Development of a Curriculum Analysis and Simulation Library 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

May, 2017

Acknowledgments

I would like to thank my committee, Professor Greg Heileman, Doctor Chaouki Abdallah, Doctor Ahmad Slim, and Doctor Terry Babbitt. I would especially like to thank Professor Heileman for all of his support and guidance throughout my academic career.

Development of a Curriculum Analysis and Simulation Library 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

Higher education institutions spend much of their resources in attempts to improve student success by addressing improving instruction quality, implementing tutoring programs, and providing other services to help students. While these are all worthwhile, one area that tends to be overlooked is the structure of the curricula that are offered. When viewing curricula as data, or more specifically, a graph, it is intuitive to see how it's structure can influence a student's ability to move through it. However, there are currently no tools to analyze a curriculum's structural properties and how they might affect student success. This thesis describes a software library that was developed to address this issue by providing the ability to represent curricula in a programming environment as well as a set of tools to evaluate the complexity of curricula and simulate students moving through them. Furthermore, the application of these tools are shown through several experiments that demonstrate a negative correlation between a curriculum's complexity and student success.

Contents

List of Figures

Glossary

CASL (Curricular Analysis and Simulation Library) A software library designed to capture and analyze curricula as well as simulate students moving through a curriculum.

Discrete-Event Simulation A simulation model in which sequences of events occur in discrete time.

JSON (JavaScript Object Notation) A common data-exchange format.

Julia A programming language designed for both general purpose and scientific computing.

Structural Complexity A measure of a curriculum's complexity based solely on it's structural properties.

Instructional Complexity A measure of a curriculum's difficulty.

Curriculum Complexity A curriculum's overall complexity based on some function of it's structural and instructional complexity.

Chapter 1

Introduction

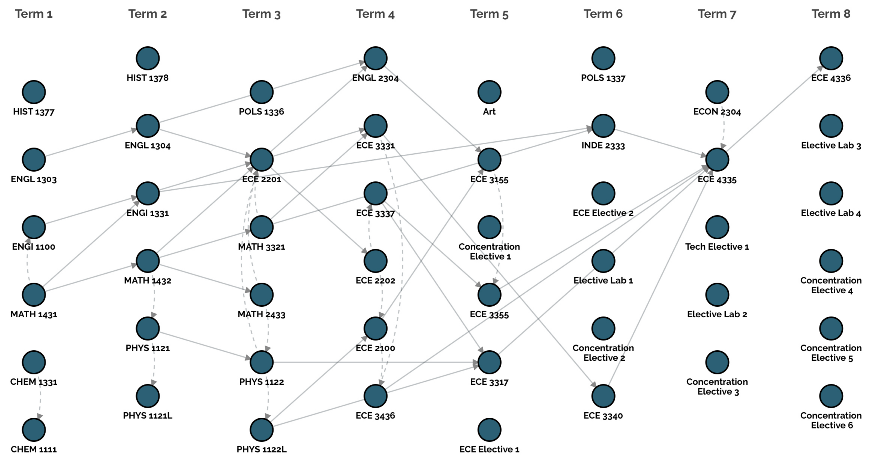
The goal of every higher-education institution is to successfully educate students with the skills and knowledge needed to enter, and succeed in the workforce. This is accomplished by offering courses with learning outcomes required for success in a given field of work. These courses are then structured 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 fields. Even more, universities are not only 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 universities spend much time, money, and research with the goal of improving 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

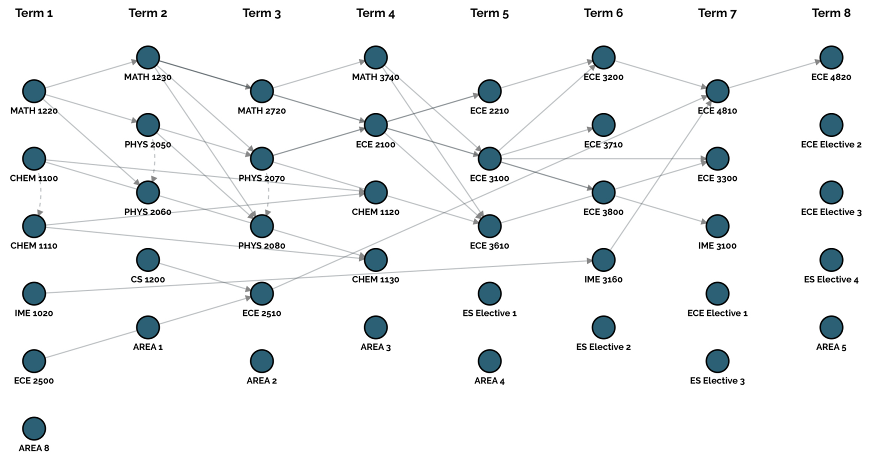
[7, 13, 4, 5, 6]. While these are all worthy pursuits that address many aspects of a student's academic career, there is one area that many universities often overlook when it comes to improving a student's journey through a curriculum: the structure of the curriculum itself.

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 program at one university will most likely be the same as that of 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. By doing so, it becomes apparent that the way in which a curriculum is structured can influence the way in which students can move through it and therefore effects the efficiency of a student. For example, consider figure 1.1. Here, you can see a comparison of two Electrical Engineering curricula from two different institutions. 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 library, the Curricular Analysis and Simulation Library (CASL) that allows for formal analytics over curricula to be done to help answer questions like the one previously posed. The library described is an open-source Julia package that allows curricula to be represented in a programming environment, provides metrics for quantifying the complexity of a curriculum. The central feature of this library is, as it's name implies, the simulation 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 graduate, 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. In addition an overview of this software library, this thesis details several experiments that make use of it to demonstrate the correlation between a curriculum's structure and student success.



(a) University of Houston



(b) Michigan University

Figure 1.1: Comparison of two Electrical Engineering Curricula

Chapter 2

Previous Work

2.1 Defining A Curriculum's Structural Complexity

Before being able to determine the effects a curriculum's structure has on the ease in which students can move through it, it is necessary to quantify the structure, or rather the complexity of a curriculum's structure. A team of researches at the University of New Mexico have done just that by introducing two metrics that attempt to capture properties of a curriculum structure that pertain to a students progressing though it [12]. As previously seen in figure 1.1, a curriculum can be represented as a graph allowing it to be subject to graph theory and complex network analysis, which serves as the basic of UNM's work. Computing this metric, which is simply called the *structural complexity* of a curriculum, begins by assigning each course a value, called its *course cruciality*, which signifies how important the course is in the curriculum. This value is in turn comprised of two other measures: the course's *blocking factor* and it's *delay factor*.

A course's delay factor is defined as the number of nodes (or courses) on the longest path that passes through the given course. An example can be seen in figure

2.1. In this figure, course A has a delay factor of three as the longest path that passes through it contains three courses: A, B, and D, as opposed to the short path with length two that contains courses A and C. Course B and D share course A's delay factor of three, while course C only has a delay factor of two.

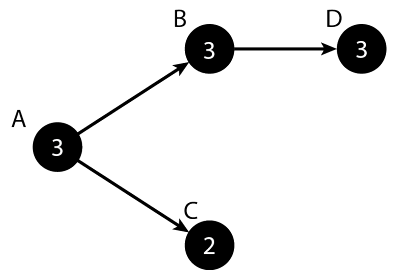


Figure 2.1: Delay factors are given for courses A, B, C, and D.

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 node i to j then n_{ij} is 1 and 0 otherwise. Then, the blocking factor of course i would be given by:

$$V_i = \sum_j n_{ij} \tag{2.1}$$

An example of blocking factors can be seen in figure 2.2. Here, course A has a blocking factor of three because it is connected (or blocks) three other courses: B, C and D. Course B blocks one course, D and course C and D have a blocking factor of zero because no courses succeed them.

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

$$C_i = V_i + L_i \tag{2.2}$$

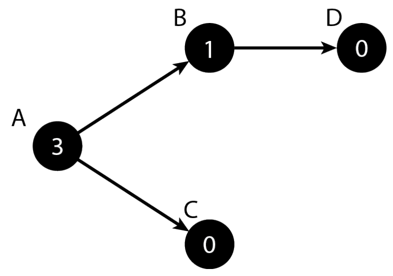


Figure 2.2: Blocking Factors are given for courses A, B, C, and D.

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

$$S = \sum_{i}^{n} C_i \tag{2.3}$$

2.2 Student Simulations

There has been much research done predicting and analyzing student performance, the majority of which use statistical analysis or machine learning models [11, 1, 3]. Although less so, there has also been work done in this area using simulations with a variety of motivations. For example, Webster [14] describes a simulation model that takes into account student enrollment, teaching resources, and financial data to run simulation to aid financial decisions. Plotnicki and Garfinke [8] use simulations to determine the best possible schedule of courses that allow the greatest number of students to easily move through the curriculum and similar work by Schellekens et al. [10] observed the effects of giving students more flexibility within a program via simulation.

While all of this research shares some similarity with CASL simulations, the work that it relates to the most was conducted at San Fransisco State University using simulations to observe how changes made to a curriculum affect student performance [9]. Saltzman and Roeder modeled students flowing through SFSU's College of Business as a discrete event simulation. In their implementation, every semester students would register for courses using the school's historic data to determine course demand. Students would then be enrolled in their desired courses given that they met all course requirements and there was room. Once enrolled, a student would either pass or fail based off the course's historic pass-rate. Using this model, they determined the effects of curriculum changes such as removing courses or prerequisites.

While this method influenced the design of CASL's simulations there are some major differences. For example CASL makes different assumptions about student behavior. For example student course demand is not based on historic data. CASL also lacks a new incoming class each term, tracks more student data (such as GPA) and can be extend to make pass/fail determinations based on more than just pass-rate data and supports grade assignments. The largest is differences however, are implementation and motivation. While Saltzman and Roeder's and the others' simulations were implemented using commercial software, CASL is free and open source and does not build the curriculum into the simulation logic, but rather is flexible enough to perform simulations over any curriculum through a standard format. More than that, CASL extends beyond just simulation and provides a way to represent curriculum-related data in a programming environment that could be useful apart from simulations.

Chapter 3

Library Design & Implementation

3.1 Julia

CASL 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" [2]. 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 [2]. 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 library's various components, methods logic. Working with data formats such as JSON and CSV were especially easier that 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 it's support for parallelism allowed running many simulations much faster as they could be carried out in parallel. Topping it all off is it's built in package manager with a plethora of packages that will enable uses extend the library and simulation capabilities with powerful machine learning and statistical models.

3.2 Architecture and Implementation Details

CASL 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 library 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 are modules which are passed into the simulation method that determine how the virtual students 'perform' in each course. The library comes with one build-in module based on course pass rates, but users can develop their own to conduct simulations that suite their needs. A system diagram can be seen in figure 3.1. The next several sections will describe each component in greater detail.

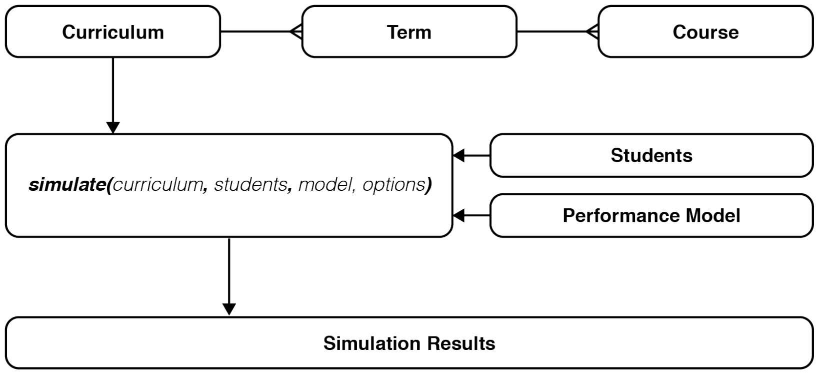


Figure 3.1: A component diagram of CASL's primary components.

3.2.1 Library Data Types

Custom Julia types serve as the foundation of the library allowing curricula and related entities to be defined a 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. A course's attributes include it's credit hours, prerequisites, co-requisites, a minimum term requirement and various complexity values, such as delay and blocking factors along with others. In addition, it also has attributes that are used by the simulation to keep track of it's state (such as which students are enrolled) as well as statistics regarding the course during the simulation such as it's simulated pass-rate, how many students were enrolled each term, grade distributions, etc. A course also has an attribute, named *model*, that has no type and can used to store any value or type for the purpose of storing a model to predict student grades. This will be explained in more detail later.

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 and 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 minimum term requirement, or both. Examples of the constructors can be seen below.

```julia
# Import the framework
using CASL

# Math 162 course with no pre or corequisite.
math162 = Course("Calculus I", 4, [], [])

# Physics course with a passrate of 90%, and math 162 as a prerequisite.
phys162 = Course("Physics II", 3, 0.9, [math162], [])

# Physics lab with Pysicis II as a corequisite
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. In addition to a list of courses a term's attributes include the total number of students that are enrolled in each of it's courses, the total number of credit hours in that term, and the number of students which failed courses in the term. The term type only has a single constructor that accepts an array of courses:

```
using CASL

# Math 162 course with no pre or corequisite.
math162 = Course("Calculus I", 4, [], [])

# Physics course with a passrate of 90%, and math 162 as a prerequisite.
phys162 = Course("Physics II", 3, 0.9, [math162], [])

# Physics lab with Pysicis II as a corequisite
phys162l = Course("Physics II Lab", 1, [], [phys162])

term1 = Term([math162])
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 library uses to perform a simulation over a curriculum. Of course the curriculum contains an ordered set of terms but additional attributes include various structural complexity measures, and average pass-rate. Also, like the course type it too has an attribute, named *stopoutModel*, that can hold any information for the purpose of storing a model to predict whether a student would stop out at the end of a term. Again, this will be explained in greater detail later.

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:

```json
{
  "terms": 2,
  "courses": [
    {
      "name": "Calculus I",
      "credits": 4,
      "passrate": 0.7,
      "term": 1,
      "corequisites": [],
      "prerequisites": []
    },
    {
      "name": "Physics II",
      "credits": 3,
      "passrate": 0.8,
      "term": 2,
      "corequisites": [],
      "prerequisites": ["Calculus I"]
    },
    {
      "name": "Physics II Lab",
      "credits": 1,
      "passrate": 0.9,
      "term": 2,
      "corequisites": ["Physics II"],
      "prerequisites": []
    }
  ]
}
```

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:

```julia
# Curriculum from term objects
myCurriculum = Curriculum("My Curriculum", [term1 term2])

```

```
# Curriculum from JSON file
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. This is possible due to a dictionary (an associative array) attribute, named *attributes*. The student's object also stores statistics for simulations such as the total number of credit hours a student has earned, the term in which each course was completed, the student's GPA, grade's received, if/when the student graduated etc. The student type only has one constructor which simply takes a dictionary of student attributes as an argument. Below is an example:

```
using CASL

# Attributes
attributes = Dict(
    'ACT' => 36,
    'HSGPA' => '3.45'
)

# Student Object
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 object is not created manually, but instead it is returned when a simulation is performed. The simulation object's attributes include a copy of the curriculum that was simulated, all simulation parameters, the number of students, graduation and stop-out rates for each term, and arrays that contain enrolled, graduated and stopped-out students.

3.2.2 Performance Modules

Performance modules are used to predict the grades that students make in their enrolled courses during a simulation. 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 simulations 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:

```
# SampleModule

function train(curriculum)
  # Loop through each course in a curriculum and train each one
  for course in curriculum.courses
    # Load some training data
    data = read_data_here("./some/location/$(course.name).csv")

    # Train a model
    # For example, obtain the params for a linear regression.
    model_params = train_model(data)

    # Store the model
    course.model[:params] = model_params
  end

  # Model for predicting stopouts
  # Here this could simply be a probability.
  curriculum.stopoutModel[:rate] = 0.1
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:

```
# SampleModule
function predict_grade(course, student)
  # Get the parameters from course
  params = course.model[:params]

  # Construct model based on the params
  model = build_model(params)

  # Construct a feature sample from student
  sample = [student.attributes[:ACT], student.attributes[:GPA]]

  # Predict grade
  grade = predict(model, sample)

```

```
   return grade
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 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:

```
# SampleModule
function predict_stopout(student, currentTerm, model)
 # Chance of stopping out
 chance = model[:rate] - currentTerm*0.01

 # Determine stopout
 return rand() <= chance
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 CASL 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 a 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:

```
 # Predicts whether a student will pass a course using the course's passrate
# as a probability.

module PassRate
    # Train the model
    function train(curriculum)
        for course in curriculum.courses
            model = Dict()
            model[:passrate] = course.passrate
            course.model = model
        end

        curriculum.stopoutModel[:rates] = [0.0838, 0.1334, 0.0465, 0.0631, 0.0368,
            0.0189, 0.0165] * 100
    end

    # Predict grade
    function predict_grade(course, student)
        roll = rand()

        if roll <= course.model[:passrate]
            return 4.0
        else
            return 0.0
        end
    end

    # Predict stopout
    function predict_stopout(student, currentTerm, model)
        if currentTerm > 7
            return false
        else
            roll = rand(1:100)
            return roll <= model[:rates][currentTerm]
        end
    end
end
```

3.2.3 Simulation Method

All of the components that have been described come together to simulate students flowing through a curriculum. This simulation is carried out through CASL's *simulate()* method. This method requires two required arguments: a curriculum, and an array of students. It will also accept several optional arguments: *max_credits*, *duration*, *performance_model*, 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. To tell the simulation to use a custom performance module for grade predictions, then the defined module can be passed in as the *perfor-*

mance_model argument. If no model is passed in, then the included pass-rate model is used. Finally, the *stopouts* argument is a boolean value that tells the simulation whether to include stopout behavior. If false, then students will never stop out, while a true value will use the Performance Module's *stopout()* method to un-enroll students at the end of every term. An example of how all these pieces come together to run a simulation is shown below:

```julia
using CASL

# Load the user defined performance module
# in this case called 'ProbitModel'
require("./path/to/CustomPerformanceModel.jl")
# => CustomModel

# Load the Curriculum
curriculum = Curriculum("./path/to/curriculum.json")

# Create a set of students with a random ACT and
# high school GPA based on a normal distribution
students = []
for i=1:1000
    attributes = Dict(
        'ACT' => random_act(),
        'HSGPA' => random_gpa()
    )
    s = Student(attributes)
    push!(students, student)
end

# Now a simulation can be run using the coures passrate model
sim1 = simulate(curriculum, students)
println(sim1.gradRate)
# => 0.73

# Tweak the parameters and use the custom performance model
sim2 = simulate(curriculum, students; duration=10, max_credis=19, performance_model=
    CustomModel, stopouts=false)
println(sim2.gradRate)
# => 0.85
```

Chapter 4

Simulation Details

4.1 Discrete Event Simulations

A 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.
- **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. 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. Given the nature of this process, modeling students moving through a curriculum as a discrete-event simulation makes sense.

4.2 Simulation 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 decision that was the most influential in making these assumptions 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-school, or transfers, students in the simulation begin with a clean slate. They are also treated as full-time students with no notion of ṕart-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 is 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.

- 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.

4.3 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.

- 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.

4.3.1 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 students 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, 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. They 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 5

Application in Curricular Analytics

5.1 Charactering Curriculum Complexity

Before describing the contribution the set of tools given by CASL makes to curricular analysis, it would help to have a more formal definition of curricula complexity. Consider a curriculum C. The complexity of C, denoted Ψ_C can be characterized as some function of two components: structural complexity and instructional complexity, denoted by α_C and γ_C, respectively:

$$\Psi_C = f(\alpha_C, \gamma_C)$$

A curriculum's structural complexity, α_C, is determined solely from the a curriculum's structural properties and is therefore also a function of it's graph, G_C:

$$\alpha_C = g(G_C)$$

A curriculum's instructional complexity, γ_C, quantifies the difficulty of the curriculum and can therefore be represented as a function, $h()$, of each course's inherent difficulty:

$$\gamma_C = h(\text{course difficulties})$$

Given these definitions, curriculum C's complexity can be expressed as:

$$\Psi_C = f(g(G_C), h(\text{course difficulties}))$$

There are many factors that might contribute to a curriculum's difficulty, most of which might be impossible to quantify therefore making it impossible to perfectly characterize γ_C. Likewise, it can also be difficulty to perfectly quantify α_C. Even though the structure of a curriculum lends itself to being quantified more easily than instructional complexity, it is unknown what structural properties to take into consideration, making $g()$ difficulty to characterize.

This is where CASL can be very useful. It can aid in better quantifying α_C by providing several measures of structural properties and, through simulation, see which ones correlate more with success. Similarly, it can also approximate $h()$ through models that can predict success in a course based off student characteristics. Finally, through simulations, a connection between a curriculum's complexity and student success can be observed without knowing Ψ_C exactly.

5.2 Structural Complexity and Student Success

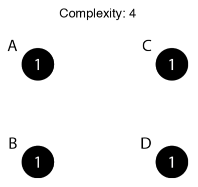
One of the primary motivations for the development of CASL demonstrate a relationship between the structure of a curriculum and the ease in which students can move through it. More specifically, that there exists a negative correlation between, Ψ_C and student success. Several experiments were designed to test this hypothesis by simulating students moving though a curriculum using the CASL's default pass-rate module (students are randomly determined to pass a course based on it's pass-rate). This provides 'best-case' completion rates which can be compared against a curriculum's computed structural complexity to determine if such a correlation exists. Here the following assumptions are made: (1) Student success will be quantified as simulated completion rates, (2) α_C will be quantified as the sum of a curriculum's delay and blocking factors, and (3) $h()$ is approximated as the average passrate of a curriculum's courses. Therefore, a curriculum's complexity can be expressed as:

$$\Psi_C = f(C_{delay} + C_{blocking}, \text{average passrate})$$

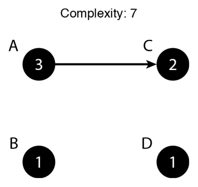
As a start, the correlation between student completion rates and Ψ_C is examined in very simple curricula which can be seen in figure 5.1. These curricula, consisting of two terms with two courses each make up fundamental patterns that can be found in actual curricula and span of range of complexities making them good candidates to begin to understand how structure affects students moving through a curriculum.

For each of these curricula, a series of one hundred simulations were run with one hundred students each where each simulation had the following parameters: (1) Every courses had a 50% pass-rate, giving each student a one in two chance of passing the course. (2) Each student could take up to three courses a term. (3) A duration of four terms. The results of each set of simulations were averaged together and can be seen in table 5.1

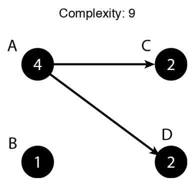
These tables give insights into how students progress through a curriculum by showing the percentage of students that have passed each course at the end of each semester. For example, looking at table 5.1a, 75.2% of students have passed Course A after the second term. The row at the bottom shows the percentage of students that have completed all courses (ie graduated) at the end of each term. As expected, the least complex curriculum had the highest completion rate after four terms, while the most complexity curricula had the lowest completion rate. It is also interesting,



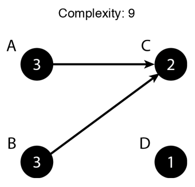
(a) Curriculum One



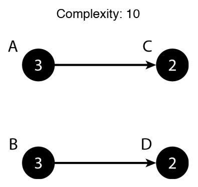
(b) Curriculum Two



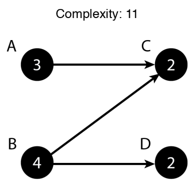
(c) Curriculum Three



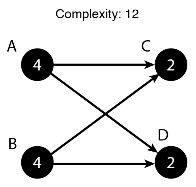
(d) Curriculum Four



(e) Curriculum Five



(f) Curriculum Six



(g) Curriculum Seven

Figure 5.1: Fundamental Curricula Structures

though not surprising, that the two curricula with the same complexity had roughly the same fourth term completion rate even though their structures were different.

To expand upon the previous experiment similar simulations were conducted over slightly more complex curricula with an additional term consisting of two courses for a total of 6 courses. Even with the addition of only two courses, the number of structures greatly increases providing a much larger curricula set with a wider range of complexities. In total, 256 curricula were systematically generated with the three terms, six courses specification. For each generated curricula, simulations similar to those done with the simple curricula were carried with slightly different parameters: (1) Every courses had a 80% pass-rate. (2) Each student can take up to three courses a term. (3) A duration of five terms. The results of these simulation can be seen in figure 5.2.

Table 5.1

(a) Curriculum One

Course	Term 1	Term 2	Term 3	Term 4
A	49.62	75.2	87.33	93.73
B	50.64	75.48	87.64	93.97
C	49.98	75.62	87.98	94.14
D	0.0	44.1	71.89	85.77
Completion Rate	0.0	20.72	49.5	71.63

(b) Curriculum Two

Course	Term 1	Term 2	Term 3	Term 4
A	51.19	75.31	87.23	93.48
B	49.94	74.71	87.31	93.45
C	0.0	25.5	51.27	69.39
D	49.3	75.2	87.31	93.59
Completion Rate	0.0	14.35	39.21	60.41

(c) Curriculum Three

Course	Term 1	Term 2	Term 3	Term 4
A	49.96	74.78	87.43	93.38
B	49.68	74.57	87.28	93.95
C	0.0	25.59	50.51	69.2
D	0.0	24.61	49.64	68.83
Completion Rate	0.0	8.88	30.01	52.57

(d) Curriculum Four

Course	Term 1	Term 2	Term 3	Term 4
A	49.98	75.12	87.93	93.92
B	49.76	75.44	87.37	93.73
C	0.0	12.35	34.7	55.21
D	49.98	75.4	87.79	94.33
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

(f) Curriculum Six

Course	Term 1	Term 2	Term 3	Term 4
A	49.92	74.86	87.8	93.74
B	50.63	75.91	88.05	93.85
C	0.0	24.45	50.1	68.89
D	0.0	12.22	34.83	56.31
Completion Rate	0.0	5.92	23.77	45.39

(g) Curriculum Seven

Course	Term 1	Term 2	Term 3	Term 4
A	50.31	75.53	88.12	93.91
B	50.82	75.56	87.55	93.83
C	0.0	12.6	34.16	55.65
D	0.0	12.89	35.47	56.29
Completion Rate	0.0	6.61	22.15	42.24

Again, the results imply a correlation between structure and completion rates. To more formally characterize this correlation, linear regression was performed over the data, the details of which can be seen in table 5.2 and is visualized by the red line in figure 5.2. The R-Squared value is a 94%. Based on this regression, for every point of complexity added, the fith-term completion rate will decrease by 0.7%, which is fairly significant. While these results are good and do indicate a negative correlation, they are not perfect. Unlike the results from the four-course curricula where the two with the same complexity resulted in the same completion rates, introducing two more courses also introduces more variance between completion rates of same-complexity

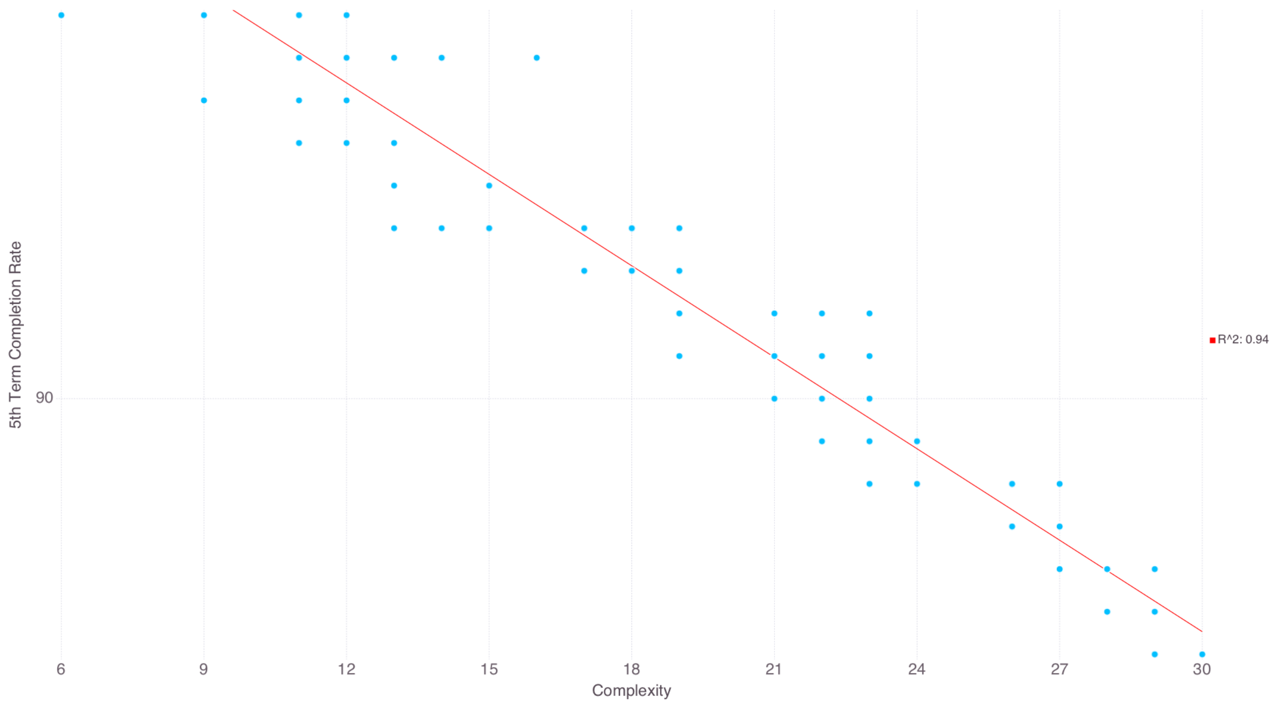


Figure 5.2: A plot of complexity vs 5th term graduation rates for six course curricula.

curricula.

Table 5.2: Six-Course Curricula Linear Regression Results

Coefficient	Estimate	Std Error	Z Value	Pr(>|z|)
(Intercept)	105.995	0.235982	449.166	<1e-99
Complexity	-0.715459	0.0113234	-63.1844	<1e-99

The final set of experiments were conducted over real-world curricula pulled from curricula.academicdashboards.org, a web service that stores, visualizes and computes the complexity of curricula. It hosts overs 120 curricula from Universities around that country, and pulling those that have at least eight terms and between 100 and 150 credit hours results in a set of thirty eight curricula. These curricula are then simulated 50 times each with 1000 students using the parameters: (1) Every courses had a 80% pass-rate. (2) Each student can take up to eighteen credit hours a term. (3) A duration of ten terms. A plot of the results can be seen in figure 5.3.

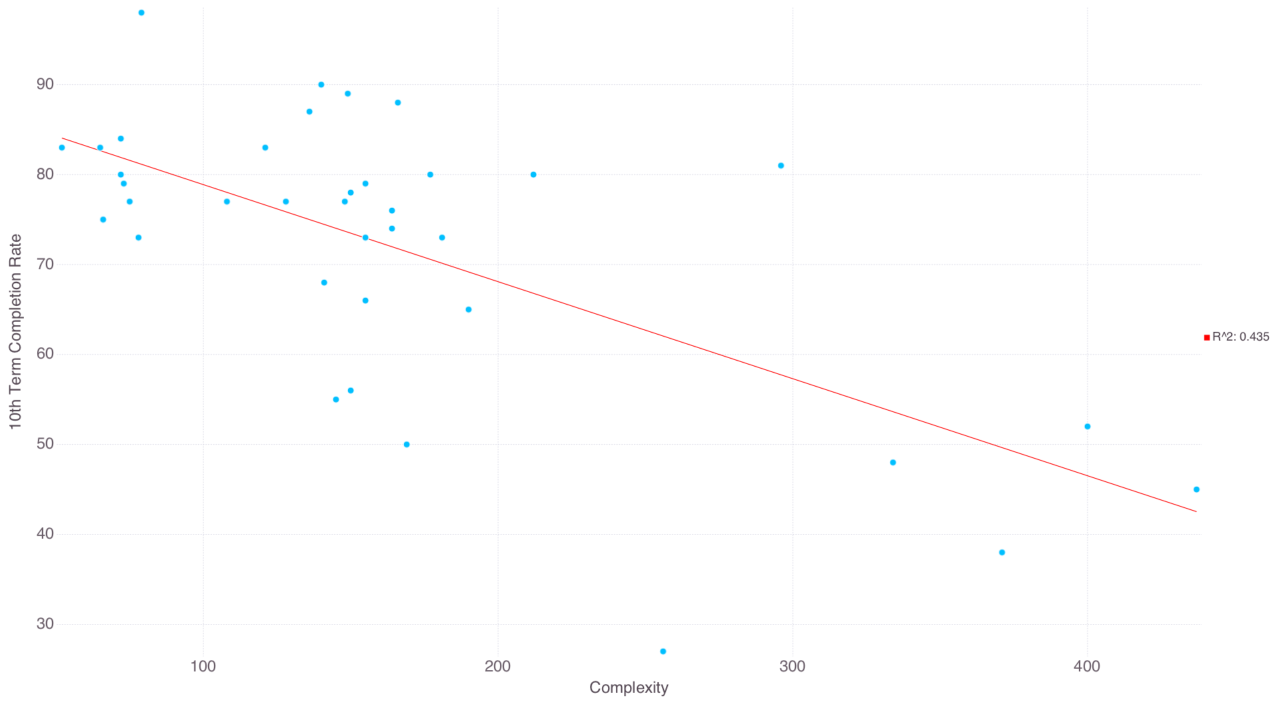


Figure 5.3: Simulation results of thirty two real-world curricula.

Like the two experiments before, an inverse relationship exists between complexity and completion rates, however here there is much more variance. Again linear regression was employed, indicated by the red line, and it is easy to see that there is not as tight of a fit to the results as there was with the six-course curriculum. This is indicated in the resulting R-Squared value of 0.435. This is fairly unsurprising given the more varied nature of the real-world curricula, however there might be other issues that affect this relationship. For example, the previous two experiments used sets of curricula that had the same number of courses and therefore the same number of credit hours. This is not the case with the real-world curricula which span a range of 100 to 150 credit hours. To try and account for this, another regression was performed with credit hours being a independent variable. The results of this regression can be seen in table 5.3 and is visualized in figure 5.4. It is easy to see that this change produces a much tighter, though not perfect, fit with a R-Squared value of 0.703.

Table 5.3: Real-World curricula regression results using complexity and credit hours as variables.

Coefficient	Estimate	Std Error	Z Value	Pr(>|z|)
(Intercept)	255.169	29.748	8.57771	<1e-17
complexity	-0.0884049	0.0156087	-5.66382	<1e-7
credit hours	-1.38212	0.250417	-5.51927	<1e-7

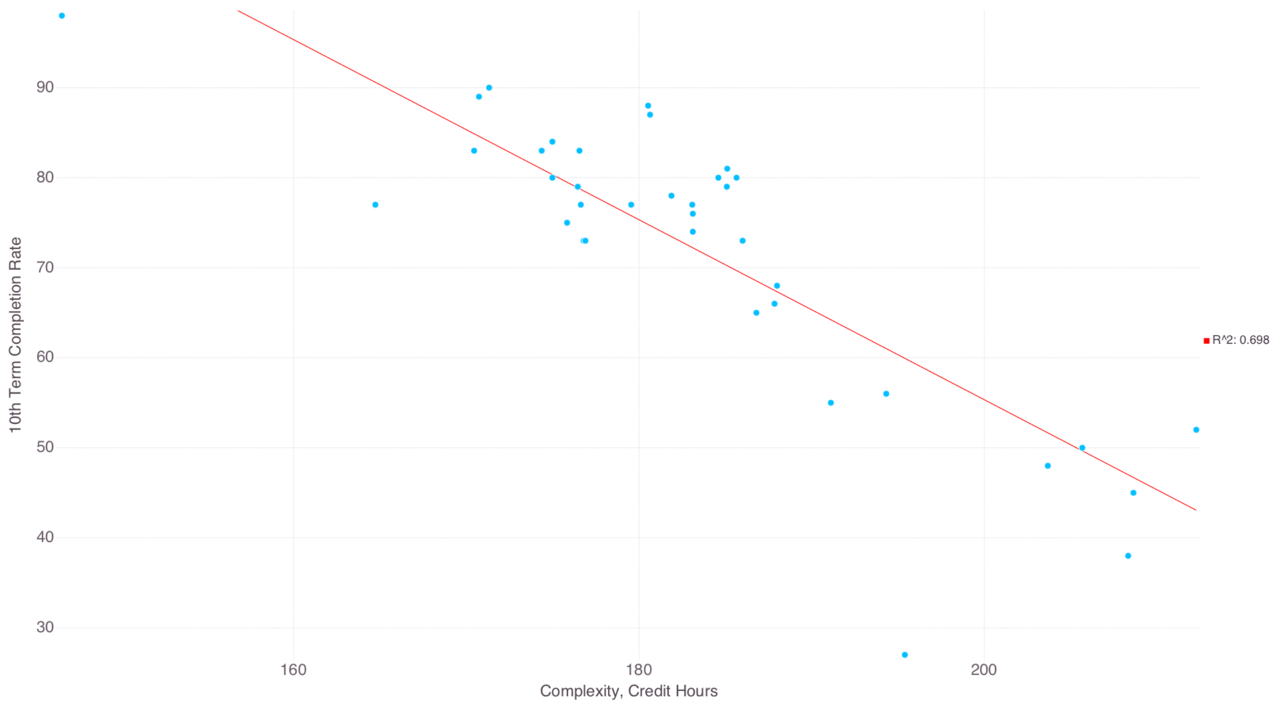


Figure 5.4: Real-World curricula regression analysis using complexity and credit hours as variables.

5.2.1 Evaluating Additional Complexity Measures

While it has been shown that both delay and blocking factors can affect students moving through a curriculum, they are far from the only way to describe the structural complexity of a curriculum, and there might in fact be better measures. This

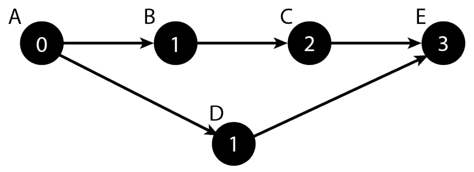


Figure 5.5: Example of course reachability.

is evident by looking at figures 5.2 and 5.3 and noticing that curricula with the same for similar complexities produce very different completion rate results and better characterizing α_C might allow for tighter correlation. As a start to exploring other structural complexity measures, four new course complexity metrics are proposed: course centrality, course reachability, free courses, and the number of prerequisite relationships which are all built into CASL.

When a student fails to pass a course, they are probably hindered from taking other courses. In cases like this students might be able to take a course that has no prerequisite (such as an elective) so that they can still make progress while retaking their failed course. Therefore the number of courses without any prerequisites in a curriculum might influence student progress. A course's reachability describes how difficult it is to get to. It is defined similarly to that of a course's blocking factor but instead of counting the number of courses that can be taken after completing the given course, it counts the number of courses that must be completed to reach the given course. A course's centrality attempted to measure how central a course is within a curriculum by counting the number of unique paths that a course appears on. Examples of these two measures can be seen in figures 5.5 and 5.6.

Given these measures the question becomes which of them or combination of them should be used to best characterize α_C. Again, simulation provides a solution. The experiments so far have tried to show how complexity influences completion

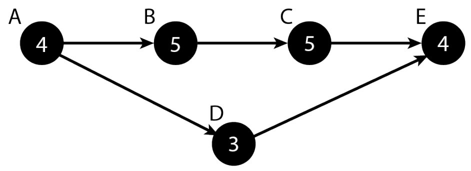


Figure 5.6: Example of course centrality.

rates by performing linear regression over the results, however this process can be reversed. It is possible to use regression to allow completion rates to influence the right characterization of structural complexity. A good way to determine the best way to define structural complexity would be to find the combination of structural measures that correlates the best with simulated graduation rates. To do this, linear regression was performed for every possible combination of structural measures as the independent variables and then find the combination that provides the highest R^2 value.

Unsurprisingly, the more features that were used produced better correlations. The combination that produced the best result was credit hours, delay+blocking, centrality, reachability, and the number of prereq relationships which resulted in an R^2 value of 84%. More specifically, based off the results, the optimal structural complexity can be defined as such:

$$\alpha_C = 1.3credits_C - 0.26(delay_C + blocking_C) + 0.03centrality_C + 0.72reachability_C - 0.44prereqs_C$$

Figure 5.7 visualizes these results.

5.3 Instructional Complexity and Student Success

The previous experiments have all focused on the affects of structural complexity, but instructional complexity, γ_C, is also important in defining the overall complexity of a

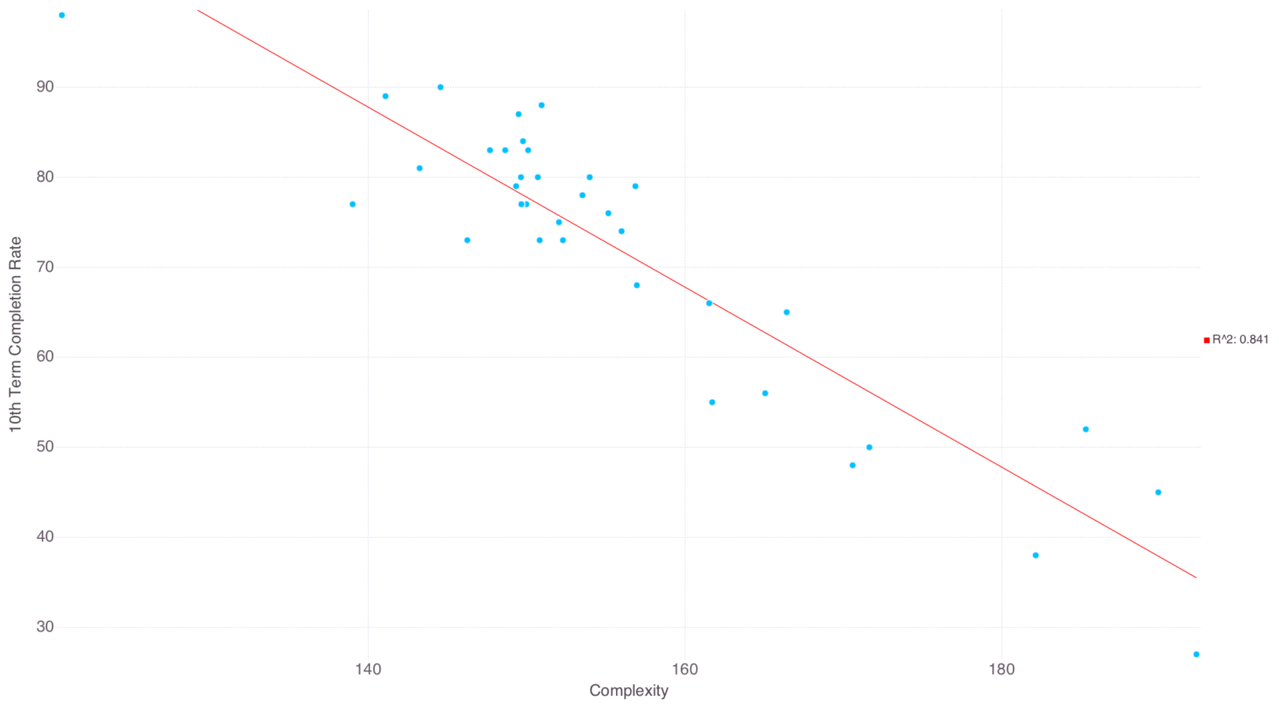


Figure 5.7: Regression analysis plot using complexity, reachability, centrality, credit hours, and number of prereqs as independent variables.

curriculum. In fact, not only does a curriculum's instructional complexity influence student success, but it also has effect on how structural complexity influences success. To demonstrate this, simulations using real-world curricula were performed with each curricula having the same passrate over a range of passrates from 0% to 90%. For the results at each pass-rate value, linear regression was used to determine the correlation between each structural complexity measure and completion rates. The results are plotted in figure 5.8.

In this figure, it is evident that the structural complexity measures correlate differently for different curriculum average-passrates. This makes sense as the structural of a curriculum that is impossibly difficulty doesn't matter if students cant pass any of it's courses and likewise also doesn't matter if students always pass their courses.

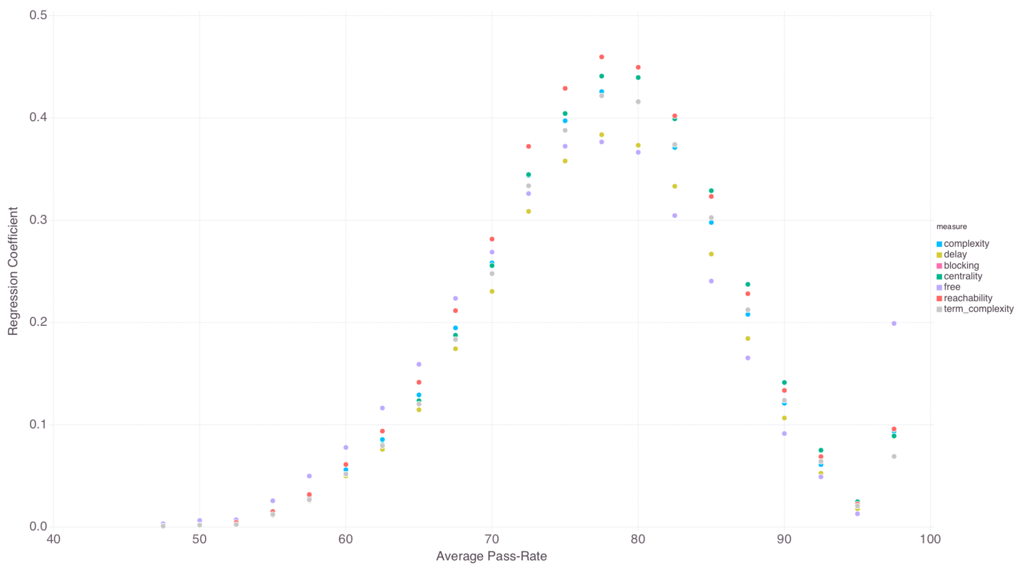


Figure 5.8: Graph showing how the completion rate of the Computer Engineering curriculum at UNM changes with respect to course pass-rates.

Therefore, instructional complexity determines how important a curriculum's structure is.

While it is obvious that a curriculum's difficulty largely affects how students progress through it, simulations can show the exact nature of this relationship by simulating the same curriculum (UNM's Computer Engineering program) several times while varying the curriculum's course difficulty. While there is a linear relationship between structure and success, the influence of instructional complexity looks very different which can be seen in figure 5.9.

5.4 Course Difficulty Sensitivity Analysis

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-

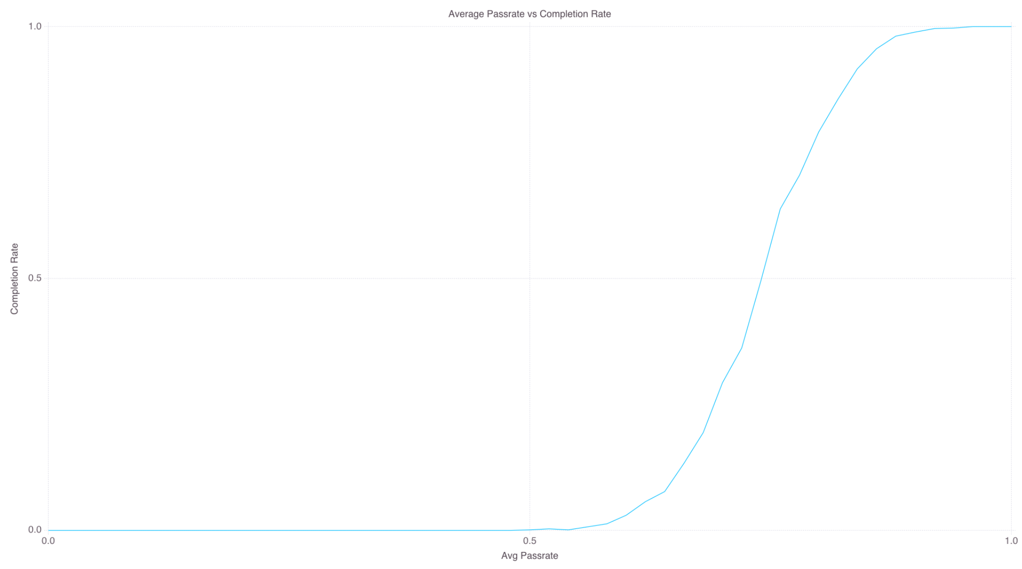


Figure 5.9: Graph showing how the completion rate of the Computer Engineering curriculum at UNM changes with respect to course pass-rates.

rate of critical courses can increase student graduation rates. So then the question becomes which courses should receive the 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 5.4, 5.5, and 5.6 show the results of this analysis - specifically the top five courses in which an increase in pass-rate had the greatest affect on eighth-term completion rates. 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.

Course	Original Pass-rate	Increased Pass-rate	Cruciality	Increase in Completion Rate
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

Table 5.4: 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

Table 5.5: Mechanical Engineering Sensitivity Analysis

Course	Original Pass-rate	Increased Pass-rate	Cruciality	Increase in Completion Rate
MATH 162	0.7585	0.9861	19	0.0609
MATH 163	0.7502	0.9753	18	0.0554
PHYC 160	0.7926	1	14	0.0287
MATH 264	0.8399	1	5	0.0251
ECON 105	0.7904	1	1	0.0239

Table 5.6: Computer Engineering Sensitivity Analysis

Chapter 6

Conclusion/Future Work

The software library described in this paper provides a good first step towards providing tools to perform analytics over curricula and student data though there is still plenty of work that can be done both on the features of the library itself as well as the analysis enabled by it. First, the library could be improved with polish such as better error-handling, generators for module extensions, and built in documentation. The feature set could also be expanded especially in terms of simulation customization. It has been explained that the simulation operates under several assumptions that were made intentionally, however some of these assumptions could be lifted. For example, courses can be set to only be offered only in certain terms, or they could have class-size limits. Furthermore, support could be added for more robust student-enrollment procedures. In this same vein, more out-of-the-box modules could be included to predict grades using a variety of machine-learning and statistical methods. These kinds of improvements of the simulation procedure enable more complex student behavior which could lead to making it a more powerful prediction tool.

In terms of the analytics work enabled by the library, there is always room for improvement in the kinds of measures that can be devised to quantify the structural complexity of curricula. While a perfect measure, or set of measures that perfectly correlates to both simulated and real-world completion rates probably doesn't exist, there might still exist measures that capture structural complexities better than the ones described here. Furthermore there is also plenty of work to be done analyzing instructional complexity, or the difficulty of curricula as well. This work took a back seat to analysis of curricula complexity, but is a wide-open area of research. Using the library to simply focus on individual courses, rather than a whole curricula, especially with regards to grade-depiction could yield valuing insights. Combining this with more configurable simulations could produce accurate predictions and what-if analysis.

In addition to this work, CASL could also benefit from a GUI implementation. Given that there is already a community using the tools provided Curricular Dashboards and uses a near-identity format for representing curricula, it would be a good fit. Although it would be somewhat limited, providing an interface for performing simulations over curricula that has been uploaded would be powerful and provide a simple way for users, especially those without programming experience, to benefit from the library and gain more insights into their curricula - with the hopes that this insight might prove useful in decision making that could positively impact student success.

References

[1] P. M. Arsad, N. Buniyamin, and J. l. Ab Manan. Neural network and linear regression methods for prediction of students' academic achievement. In *2014 IEEE Global Engineering Education Conference (EDUCON)*, pages 916–921, April 2014.

[2] J. Bezanzon, S. Karpinski, V. Shah, and A. Edelman. Julia: A fast dynamic language for technical computing. In *Lang.NEXT*, Apr. 2012.

[3] C. F. Goenner and S. M. Snaith. Predicting graduation rates: An analysis of student and institutional factors at doctoral universities. *Journal of College Student Retention: Research, Theory & Practice*, 5(4):409–420, 2004.

[4] M. S. Hunter and E. R. White. Could fixing academic advising fix higher education?. *About Campus*, 9(1):20–25, 2004.

[5] K. P. King. Identifying success in online teacher education and professional development. *The Internet and Higher Education*, 5(3):231–246, 2002.

[6] W. C. Lewallen. Early alert: A report on two pilot projects at antelope valley college. 1993.

[7] 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.

[8] W. J. Plotnicki and R. S. Garfinkel. Scheduling academic courses to maximize student flow: A simulation approach. *Socio-Economic Planning Sciences*, 20(4):193–199, 1986.

[9] R. M. Saltzman and T. M. Roeder. Simulating student flow through a college of business for policy and structural change analysis. *Journal of the Operational Research Society*, 63(4):511–523, 2012.

[10] A. Schellekens, F. Paas, A. Verbraeck, and J. J. G. van MerriÃńnboer. Designing a flexible approach for higher professional education by means of simulation modelling. *The Journal of the Operational Research Society*, 61(2):202–210, 2010.

[11] A. Slim, G. L. Heileman, J. Kozlick, and C. T. Abdallah. Predicting student success based on prior performance. In *2014 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*, pages 410–415, Dec 2014.

[12] A. Slim, J. Kozlick, G. L. Heileman, and C. T. Abdallah. The complexity of university curricula according to course cruciality. In *2014 Eighth International Conference on Complex, Intelligent and Software Intensive Systems*, pages 242–248, July 2014.

[13] 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.

[14] T. Webster. Cost analysis and its use in simulation of policy options: The papua new guinea education finance model. *International Review of Education / Internationale Zeitschrift fÃijr Erziehungswissenschaft / Revue Internationale de l'Education*, 43(1):5–23, 1997.