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, 2016

Dedication

Dedication goes here.

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, 2016

Abstract

Abstract goes here

Contents

List of Figures

Glossary

Discrete-Event Simulation JSON

Chapter 1

Introduction

Chapter 2

Background

Chapter 3

Previous Work

Chapter 4

Student Flow Simulation Design

4.1 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 ellapsed 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 que 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 statitics based on the events that have occured.

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 instituion, much more so than continuous-time simulations and other analytical models; therefore, the student-flow simulation is modeled as a discrete-event simulation.

4.2 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

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 arragement 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 corequisite 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 4.1.

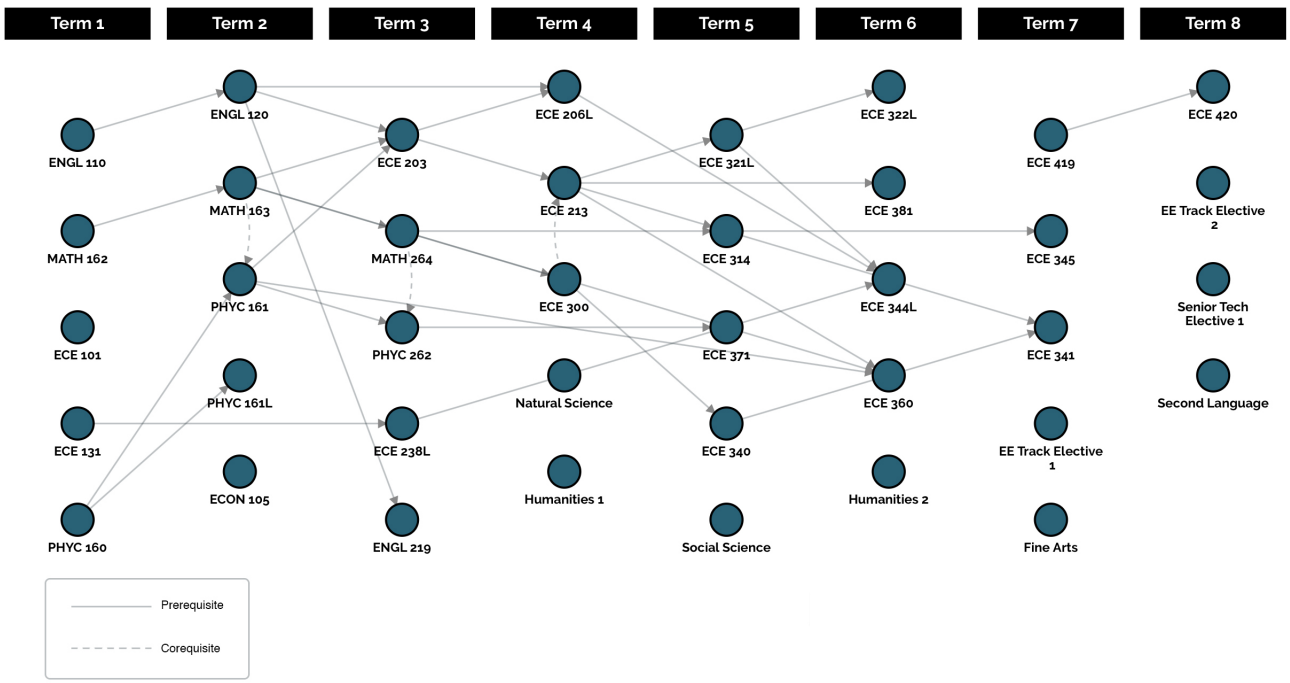


Figure 4.1: Visual representation of the electrical engineering curriculum at UNM.

4.3 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 eratic 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 wouldnt have been able to, and various other actions that would otherwise not be considered ťypical' 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 decions made, was to design the student flow simulation with the primary goal of performing analytics over curricula and observing the effects their properities 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 decsions:

- 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 fullt-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 disenrolled permantly. They do not re-enroll.

- Students will only register for courses within the specified curriculum. The simulation is only aware of the courses specificied 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 amoung 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 wasnt 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 difficuly effect outcomes which provides insight into how they might be improved.

4.4 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 semster. 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.

Statistics.

4.4.1 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 coures 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, corequisties, 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 reigister 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 semster. These events are then repeated until there are no students enrolled, or the simulation clock reaches a time set by the user.

Chapter 5

Simulation Framework Implementation

5.1 Julia

The curricular flow simulation is implemented in the Julia programming language. Julia is an open source, high-level language developed for the purpose of numerical and scientific computation while also being useable as a general purpose language. It is dynamically typed and has a syntax similar to that of ruby or python and uses a LLVM-based just-in-time (JIT) compiler which allows it to reach performance close to that of C and, in most cases, speeds faster than that of R or Matlab. It is for this reason that it was the language chosen to implement the student flow simulation. Its implentation makes it powerful and effecient for statistical and machine learning tasks, while its syntax and general purpose capabilities made it simple to implent the control flow of the simulation as well as work with data formats such as CSV and JSON, which is difficult to do in languages such as MATLAB.

Other features of Julia are multiple dispatch, support for modules, a type system,

support for parallelism, and metaprogramming facilities. Multiple dispatch allows for polymorphis by allowing multiple definitions of the same function based on the function arguments. The type system made implenting and capturing statistic for the various componets (studens, courses, etc.) simple, and its parallelism made running many monte-carlo simulations much faster. Another notable feature that played a part in the selection of Julia is its built in package manager in addition to a thriving community of developers contributing great, open-source machine-learning, statistical analysis, data-handling, and other genral-purpose libraries. These include regression tools, neural network libraries, web servers, plotting libraries, and many more.

5.2 Architecture and Implementation Details

The student flow simulation is designed as a Julia module that provides users with the basic components for carrying out simulations on their own curricula. Although the simulation is the primary functionality that the module provides, it can also be thought of as a light framework that provies building blocks for other types of analysis over curricula. These basic components consist of custom Julia types that represent the entities used within the simulation (students, courses, ect.) and at its core, a Julia function that implements the simulation control flow previously described.

The simulation also empores a modular design by separating out the logic that determines how students enroll in courses and the logic that determines how students perform in those courses. This allows users to plug in their own Julia modules that implement grade assignment/prediction models once students are enrolled in their courses. So asside from simulation, this framework can be used as a tool to make grade predictions and when these predictions are chained together, it becomes a powerful simulation tool.

5.2.1 Data Types

The various entities that are represented in the simulation are captured as custom Julia types. These are the building blocks that along with being used for simulations, can stand alone to aid in other types of academic related analtyics. These types closely resembe structures in C. Although they are not true objects as they do not have associated methods, these types do have attributes that represent the associated entity's characteristics. These attributes are also used to keep track of neccessary information during the simulation, as well as record statistics pertaining to the represented entity as the simulation is carried out. These types are: Student, Curriculum, Term, Course, and Simulation.

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 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.

cruciality The course's cruciality.

prereqs An array of courses that are considered prerequisites.

coreqs An array of courses that are considered corequisites.

postreqs An array of courses that the courses is a prerequisite or corequisite to.

termReq A courses minimum term requirement. For example, if this value is 2, then the courese 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, empies it.

passrate The course's actual, real-world passrate. 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% passrate.

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

Like all types in Julia, a course object is istantiated 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 corequisites with one of the constructors accpeting only these arguments. The others constructors allow more informaiton to be provided, specifically a passrate, a termReq, or both. Examples of the constructors can be seen below.

```
# 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. It has the follwing 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.

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:

```
# 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 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 convient 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.

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 passrates. Can be used as a naive measure of difficulty.

stoupoutModel A dictionary that stores the model that predicts student droupouts.

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 useable, is that it requires the term objects, and therefore course objects to already be istantiated. This can be a tedius and time consuming and is not a very good way to store curricula in a extensible format. To address this probelm, a JSON file format was developed as a straigtforward way to define a curriculum that is easier to create, more portable, as well as easeir 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 follwing 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 *corequisites* which is an array of strings corresponding to the names of course's corequisites. An example can be seen below:

```
{
  "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": [],
```

```
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 convienient 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 characterisitics 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:

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*3) or 9.9points.

gpa The students 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 wheter 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 consist 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 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:

```
# 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 indiviual piecies 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.

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 indicies 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 pediction module. If no module is passed in then a default is used. It has a single constructor shown below:

```
# A simulation object that uses the default grade prediction module.
sim_default = Simulation(myCurriculum)

# A simulation object that uses a custom grade prediction module (ProbitModel).
sim = Simulation(myCurriculum, model=ProbitModel)
```

5.2.2 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 themsleves. 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 cary 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 begnis and performans 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 the framework will be used. This default model simply uses the passrate 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:

```
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,
               0.0189, 0.0165] * 100
14     end
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
```

5.2.3 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 remails 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:

```
# Import the framework
using CurriculumFlow

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

# 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

# Create the simulation object
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 6

Application in Curricular Analytics

6.1 Curricular Complexity

Before using simulation to evaulate and analyse curricula, it would be useful if there was a metric that describes how complex a curriculum is. Luckily, such as metric has been devised by researches at UNM. This metric is called the *structural complexity* of a curriculum. As previously seen in Figure 4.1, a curriculum can be viewed as a graph, with nodes representd 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 it's *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 6.1. 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 oppsed 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.

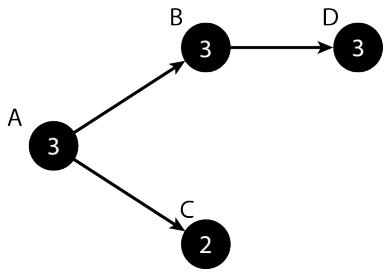


Figure 6.1: Delay factors are given for courses A, B, C, and D.

A course's blocking factor is defined as the number of courses that are blocked from being taken if the given courses 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

$$V_i = \sum_j n_{ij} \tag{6.1}$$

An example of blocking factors can be seen in Figure 6.2. Here, course A has a blocking factor of three because it is connected (blocks) three other courses: B, C and D. Coure B blocks one course, D and courses C and D have a blocking factor of zero because no coures succeed them.

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

$$C_i = V_i + L_i \tag{6.2}$$

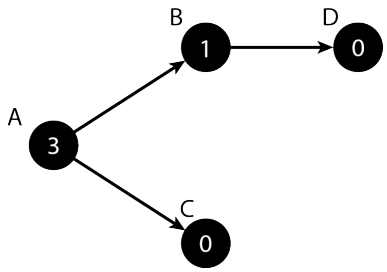


Figure 6.2: Blocking Factors are given for courses A, B, C, and D.

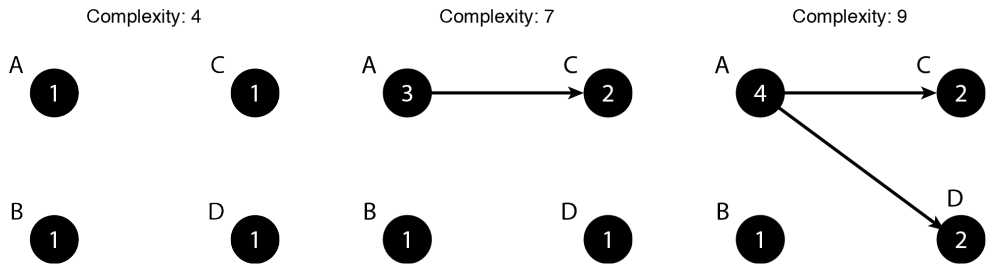
A curriculum's complexity, S is then given by the sum over all course crucialities:

$$S = \sum_{i}^{n} C_i \tag{6.3}$$

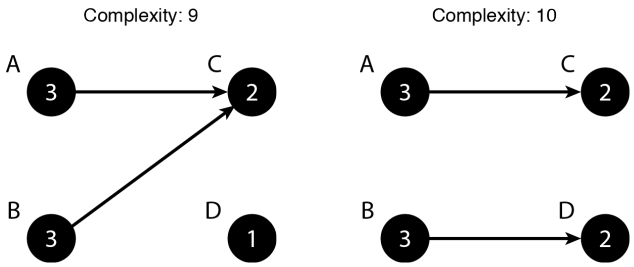
While it is far from the only factor for student success, it is intuitive to see that a curriculum's structural complexity can impact the time in which it takes students to move through a program. Students that fail a course with a high blocking factor will be unable to take needed courses the following 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.

6.2 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 extreamly simple curricula seen in 6.3. These curricula, consisting of two terms with



(a) Curriculum One (b) Curriculum Two (c) Curriculum Three



(d) Curriculum Four (e) Curriculum Five

Figure 6.3: Basic Curricula

two courses each make up common patterns that can be found in actual curricua and span a range of complexity values making them good candiates 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 corse's passrate. Every course has a passrate 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 6.1.

Table 6.1

(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

These tables show the percentage of students that have passed each course at the end of each semester. For example in Table 6.1a, 75.2% of students had passed Course A after the second semester. The row at the bottom then shows the percetage 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, a web service that stores, visualizes, and computes the complexity of curricula. Each curriculum used in the experament 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 passrate of %80. The results of which can be seen in Figure 6.4.

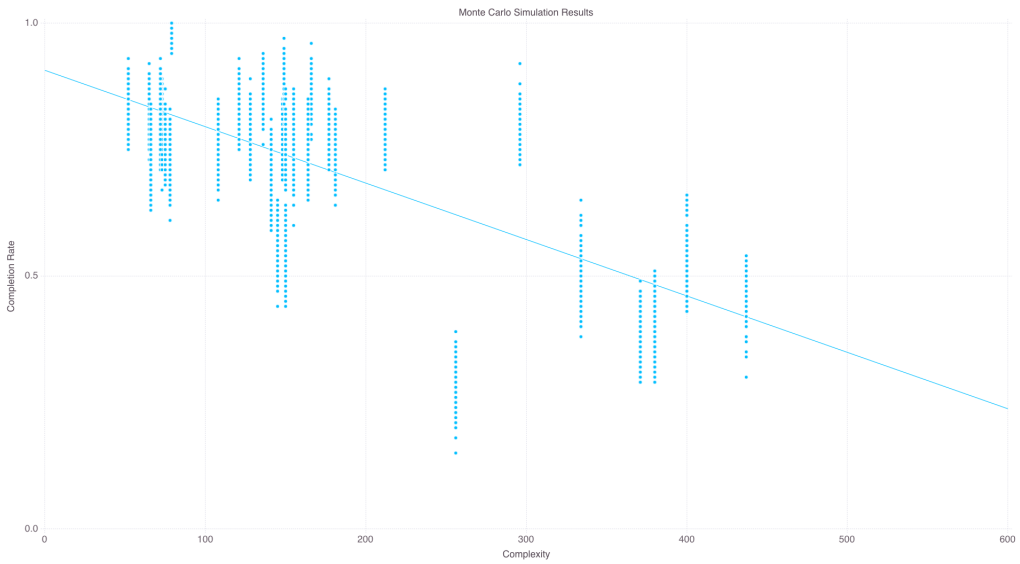


Figure 6.4: Simulation results of thirty five real-world curricula

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 6.4, curriculum can varry 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.

6.2.1 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 difficuluty of the curriculum,

which in term will be defined by the difficululty of the courses that make it up. Defining how difficuluty 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 passrate 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 difficuluty of a curriculum's courses would greatly affect student performance. Although not a realistic scenario, Figure 6.5 illustrates this point.

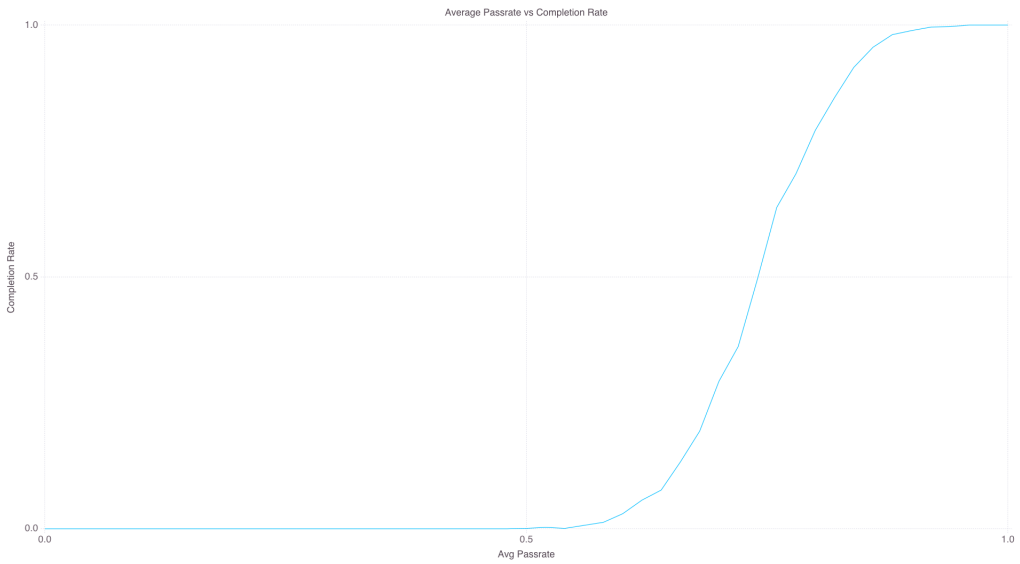


Figure 6.5: Graph showing how the completion rate of the Computer Engineering curriculum at UNM changes with repect to course passrates.

6.2.2 Probit Performance Model