5-2020

# Curriculum Optimization via Activity-on-Node Network Modeling

Caroline Rhomberg
*University of Arkansas, Fayetteville*

# Curriculum Optimization via Activity-on-Node Network Modeling

This thesis was submitted in fulfillment of the
requirements for honors distinction for the degree of
Bachelor of Science in Industrial Engineering

By

Caroline Rene Rhomberg
University of Arkansas
May 2020

Thesis Advisor: Dr. Kelly Sullivan
Thesis Reader: Dr. Richard Cassady

ACKNOWLEGEMENTS

First and foremost, I would like to thank Dr. Kelly Sullivan for his wisdom in guiding me throughout this project. He is extremely knowledgeable about networks and optimization algorithms. He encouraged me to think outside the box and introduced me to the astonishing applications of Graph Theory to intangible engineering systems, such as the flow of information.

I am also incredibly thankful for Dr. Sullivan's mentorship, as he was an excellent source of advice when applying to graduate schools while simultaneously completing this project. He cares tremendously about the wellbeing of his students and instills in them the best qualities of character.

I express my sincerest gratitude to the University of Arkansas Industrial Engineering department and Honors College for their support in funding this project in the fall and spring semesters, respectively. Thank you to Dr. Ed Pohl, Dr. Kelly Sullivan, and Dr. Richard Cassady for writing me recommendation letters for the Honors College Research Grant. Without your time, I would not have received the funding I did to enable this project.

To Dr. Jesse Delaney and the University of Arkansas Office of Student Success, thank you for providing me with the necessary data about student demographics and historical pass/fail rates that enable this model to incorporate real time data, increasing accuracy.

To the faculty members serving on my Thesis defense committee, thank you for your time and support in seeing this project to completion.

Last but not least, than you to my friends and family, who believed in me to accomplish this project and accommodated me moving back home in the latter part of the spring semester due to the Coronavirus Pandemic. There is a reason why I am the fifth oldest of six people in our family—so that I would have you all to look up to and keep me persevering towards my own goals.

TABLE OF CONTENTS

PROJECT SUMMARY

Universities attempt to plan curricula and advise students to have the best chance of passing and completing their degree on time. However, still, some students fail to pass courses, causing them to have to retake them and delay their graduation, and in extreme cases, withdraw from their university degree. High graduation rates have great implications for all facets of society. First, they are important to the individual because having a university degree increases quality of life and job security. Graduation rates are important to universities because they are used as a means of ranking and allocation of public funding. Graduation rates are also important to society as a whole because more people attaining a university education leads to lower crime rates, higher diversity of labor, and increased global trade.

The purpose of this study was to develop an optimization tool that iteratively decides the optimal progression through a list of courses, symbolic of a degree program. In an effort to simulate reality, the optimization tool was built to solve for the optimal progression through a subset of the Bachelor of Science in Industrial Engineering degree at the University of Arkansas. The model outputs the list of courses a student should take for a given semester in order to have the best probability of passing, and the model updates its suggestions as it learns more about the student, such as which courses the student has already completed and which courses they have failed and need to retake. To select a given semester's courses, the optimization model takes the form of a dynamic program in which the number of decision stages equals the number of courses to select for that semester. The dynamic program can be envisioned as a shortest path problem on a network in which nodes represent the states of the dynamic program, arcs represent the actions of the dynamic program, and paths correspond to a subset of courses that could potentially be selected for a given semester. The path length is related to the probability of a student passing all courses on the path and how imperative it is that the student completes those particular courses that semester, calculated from experimental data. The program then calculates the shortest, or least costly, path in the network of course decisions using Dijkstra's Algorithm. Statistics are recorded on the number of semesters it took the student to graduate.

## BACKGROUND AND SIGNIFICANCE

At first glance, it may seem that the only stakeholder in university education is the student. Actually though, it is the simultaneous interdependencies between students, educational institutions, and the nation as a whole that makes virtually everyone a stakeholder. Thus, it is important that graduation rates are maximized because of their importance to stakeholders.

For an individual, having a college degree is proven to have both tangible and intangible returns on the investment. According to a study done at Georgetown University, people who have at least a bachelor's degree earn 84% more than people without a bachelor's degree over the course of their lifetime, amounting to $32,000 per year and $1.4 million over a lifetime. [Heckler 2018]. They earn more employer-provided benefits such as health insurance and retirement plans, leading to better health and longer life expectancy [Heckler 2018]. Although difficult to quantify, a college education has positive effects on a person's aptitude and critical thinking skills [Heckler 2018]. Completing a college degree instills in a person confidence, time management skills, and work-leisure balance, all skills that increase quality of life. A college degree is not the only way, but it is the most reliable path toward financial stability and freedom.

Graduation rates are in the interest of educational institutions because of the tie to state financial support. "State financial support is increasingly being appropriated on the basis of performance—i.e., student outcomes, primarily measured by student graduation rates" [Hester and Ishitani 2018]. Many states are now requiring public institutions to disclose student performance statistics each year and prove that they are using state resources effectively. If they do not, they are at risk for lower funding, resources, and expenditures. With the transparency of rankings open to the public via heavily relied-on search engines such as U.S. News and World Report, a drop in rankings could cause prospective students to turn to other appealing universities on the basis of newer technology and funding opportunities. Graduation and retention rates are the primary factors in these rankings.

Our nation as a whole should also be invested in student performance because of its implications on the economy. In recent years the US has fallen behind other countries in college degree production [Powell, Gilleland, and Pearson 2012; Webber and Ehrenberg 2010]. This is likely a reason for countries such as China surpassing the US in industrial productivity. Greater graduation rates mean a greater diversity of skilled labor in the economy and consumer spending, boosting economic growth. Greater numbers of people having high-paying jobs that provide benefits means less people requiring government assistance, generating a bigger pool of money for public works. Comparing the United States to developing countries, many societal issues resolve themselves with increased education.

Semester course recommendation is one key area to examine how effective educational advising is. This study develops an optimization tool that models a degree curriculum as an activity network and recommends semester course decisions based on course failure rates and penalty values for not taking courses by certain times.

## KEY RESEARCH FINDINGS

### *Introduction*

This section highlights the areas of research that are most pertinent to the development and technical feasibility of the project. Although distinct areas of research, they are closely related and can be synthesized to build an inclusive predictive model.

### *Graph Theory*

In mathematics, a network is a set of objects that are connected together [Farahani et al., 2013]. The connections between the objects, or nodes, are known as edges [Farahani et al., 2013]. Networks are also known as graphs, thus constituting the basis of the branch of mathematics, Graph Theory.

Networks can be either directed or undirected. In directed graphs the movement from one node to another is restricted in direction, which can be imagined as an edge pointing from one node to other nodes. In undirected graphs, or bidirectional, movement from one node to any other node that it is connected to is valid, and vice versa. **Figure 1** shows an example of a directed network.

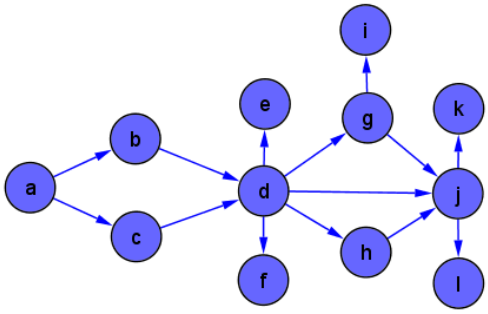


**Figure 1**: Example of directed graph [Planck 2018]

The marvel of Graph Theory lies in that networks can represent systems of all variety in the real world. For example, a network could represent a supply chain system where nodes represent stores and edges represent transportation paths. Another example could represent the World Wide Web, where nodes represent computers, and edges represent the Internet and the delivery of information to individual devices [Farahani et al., 2013]. From these contrasting examples, we see that networks can represent both physical and intangible systems.

From an implementation standpoint, a convenient way to store the information contained in a network is an *adjacency matrix*. For an unweighted network of $\boldsymbol{N}$ nodes, the network can be stored in an $\boldsymbol{N} \times \boldsymbol{N}$ matrix [Farahani et al., 2013]. Each component $a_{ij}$ represents an edge from

node *i* to node *j* [Farahani et al., 2013]. The values in the adjacency matrix consist of ones and zeros (or null values), where one indicates the existence of a connection between node *i* and node *j*, and zero (or null) means no such connection exists:

$$a_{ij} = \begin{cases} 1 & \text{if there is an edge from node i to node j} \\ 0 \text{ or Null} & \text{otherwise} \end{cases}$$

The *cost*, or length of the edge from node *i* to node *j* is not always one. The cost of a path can take on any value, and differing costs among node edges signifies that the impact of advancing from one physical location in a network or stage in a decision making process to another is not the same. For weighted networks, the adjacency matrix takes the following form:

$$a_{ij} = \begin{cases} \{x \mid x \in \mathbb{R}\} & \text{if there is an edge from node i to node j} \\ \text{Null} & \text{otherwise} \end{cases}$$

It became evident that Graph Theory is relevant to this project, as it pertains to the idea of progressing through decision stages of selecting courses for a given semester. Nodes can represent the decision to take a course or not take a course, and the progression through all nodes in a set define a path or plan for a semester schedule, from a feasible set of courses considered.

*Dijkstra's Shortest Path Algorithm*

In an unweighted network, the *costs* of advancing from node to node in a network are irrelevant; therefore the impact of every decision is assumed to be the same. However, in reality, the cost, or *value* of alternative decisions is frequently different across alternatives. Some decisions are better or worse than others. In the case of selecting courses for a semester, the cost of each decision is not the same for two key reasons. As seen in experimental data, different courses have different pass and failure rates, thus incurring unequal risks to the student when taking them. Also, due to courses serving as pre-requisites for other courses, courses also have different costs as it is more important that certain courses be completed earlier on because they correspond to more other courses.

In the scope of this problem, nodes represent states in a decision making process, or a certain number of courses selected after a certain number of stages. Edges correspond to decisions about courses. Because of this, a path through the network represents an entire selection of a number of courses. The starting node, or *source*, represents the state in which no courses have been selected, and the last node, or *sink*, represents the state in which all courses have been selected. The idea of multiple paths existing from source to sink, all of which are

comprised of edge lengths representing the cost of every sub-decision (taking the course or not), pleads for a method of finding the shortest, or least costly, path through the network.

The problem of finding the shortest path through a network turns out to be a classic problem in Graph Theory. There are many different algorithms already established. The key algorithm incorporated in this project was Dijkstra's algorithm. The algorithm was developed in 1972 by Dutch computer scientist Edsger Dijkstra [Yan 2020]. Cost in an information-based network is analogous to distance in a physical network, but the method of finding the shortest path is the same and is summarized below [Yan 2020].

1. "Distance to current vertex is zero
2. Distance to all other distances to other nodes is set to infinity
3. *S*, the set of visited vertices is initially empty
4. *Q*, the queue, initially contains all vertices
5. While *Q* is not empty do:
    a. Select the element of *Q* with the minimum distance
    b. Add *u* to the list of visited vertices, *S*
        i. For each neighbor *v* of *u* do:
            1. If the distance to *v* > distance to *u* + distance from *u* to *v*, then a new shortest path is found, and distance to *v* = distance to *u* + the distance from *u* to *v*"

Return dist [Yan 2020]

The original algorithm outputs the value of the shortest path and not the path itself [Yan 2020], however using the same principles, we can return the shortest path between two respective nodes using Dijkstra's algorithm to quantify the length of each path considered and selecting the shortest one.

The concept of finding the shortest path is especially relevant to this project. The purpose of modeling semester course selection as a network is to select the path that is the least costly to the student, which allows for the best probability of passing all courses and graduating on time. Dijkstra's algorithm is the basis for determining all decisions regarding all courses considered—either to take the course or to not.

*Dynamic Programming*

Dynamic Programming is a technique for solving optimization problems that depend on results from simpler constituent optimization problems. The optimal solution to the overall problem depends on the optimal solution to the preceding problems.

An optimal selection of courses can be built up by drawing upon solutions to smaller course selection optimization problems. The dynamic program first proceeds by figuring out the shortest path to the sink node from every node in the last node. Then, using those solutions, it finds the shortest path to the sink node from every node in the second-to-last node. This process is repeated until the shortest path is found.

*Simulation*

To validate any predictive model, simulation is a technique that allows the model to be tested with many combinations of input. In the case of this study, the goal was to simulate students progressing through a subset of courses from the Industrial Engineering degree at the University of Arkansas following the suggestions of the optimization model. Therefore, the system required a method of solving a complete progression through the subset of courses under the model suggestions to gather performance metrics across many replications. To solve this, simulation of an entire progression through the subset of input courses was modeled as a dynamic program, where each subproblem was an optimization problem of each semester's selections. In each optimization problem, the shortest path through the network represented a decision about each individual course considered for a semester. From this, information was extracted regarding which courses should be taken in order to have the best probability of passing all courses and graduate on time.

## DATA COLLECTION AND PREPARATION

### *Introduction*

This section discusses internal data that was used in the project, its format, and how it was used in the construction of the model. This data was collected in the University of Arkansas Office of Student Success.

At the University of Arkansas, data is consistently collected in the form of student grade records in the Office of Student Success. The compilation of data includes individual records of courses each student takes, many different attributes about the student, and the outcome of the course. For the purpose of this project, a request was made to the University of Arkansas Office of Student Success to receive student course/grade records for all Industrial Engineering courses at the University of Arkansas. This dataset included student grade/course records for all industrial engineering courses for 839 students for the past twenty years. The data was used to derive the pass rates for two purposes in the optimization model: one, to calculate the cost to the decision of taking or not taking a course and two, to simulate a random outcome of a student passing or not passing the course, according to its pass distribution. **Table 1** shows a truncated sample of what was included in the data.

| **Course_Section_Term** | **Catalog #** | **Course Term** | **Course Grade** | **DFW** | **Major** | **HS GPA** |
|---|---|---|---|---|---|---|
| 2333_001_1133 | 2333 | 1133 | W | 1 | INEG | 3.765 |
| 2313_001_1119 | 2313 | 1119 | A | 0 | INEG | 4.335 |
| 2103_001_1139 | 2103 | 1139 | F | 1 | INEG | 3.915 |
| 3513_001_1193 | 3513 | 1193 | B | 0 | INEG | 4.025 |
| 3613_001_1153 | 3613 | 1153 | C | 0 | INEG | 3.415 |

**Table 1:** Sample of individual course-grade records

In the data received, for each student course/grade record, one of the available attributes was ‘DFW.’ This binary variable either had the value zero or one, indicating true or false. A DFW value of one signified that the student had a DFW outcome, which means they either received a D or an F, or they withdrew from the course. In each of these circumstances, it was assumed that the student had to retake the course.

In Excel, a pivot table was used to calculate the average of the binary indicator variable DFW, by course. This information was crucial input information in the generation of the semester course decision networks. A summary of six courses’ data is shown in **Table 2.**

| Course Code | DFW Rate |
|---|---|
| INEG 2103 | 0.21 |
| INEG 2313 | 0.17 |
| INEG 3613 | 0.12 |
| INEG 3623 | 0.13 |
| INEG 4553 | 0.04 |
| INEG 4683 | 0.05 |

**Table 2:** DFW rates of six courses

METHODOLOGY

*Introduction*

This section discusses the overall strategy of the construction of the model, important characteristics about it, and how it produced output.

*Modeling Course Decisions by Relationship*

When evaluating whether or not to take a course, it must be considered according to its relationships with other courses. In this study there are three types of relationships (or lack thereof) that a course may have. The first relationship type is pre-requisite. This means that one course depends on another course which must be completed and passed for the student to take the other course. The second relationship type is co-requisite. This means that a course must be taken *with or before* another course to take that course. The third relationship type is really a lack thereof and is when a course has neither a pre- or co-requisite relationship with any other courses. It is a *free* course meaning that the course is free to be taken on its own anytime within the degree program. An example illustrating the possible decisions per course relationship is shown in **Figures 2, 3,** and **4.**




**Figure 2:** Free course **Figure 3:** Pre-requisite **Figure 4:** Co-requisite

In **Figure 2**, course 1 is a free course, therefore the set of course decisions to consider is simply $\{1, \bar{1}\}$—either to take the course or not to take the course. **Figure 3** shows a similar relationship, but it does not have the same implementation. Course 4 is a pre-requisite for course 8. If the student has not already completed course 4, the set of course decisions is $\{4, \bar{4}\}$. If the student has completed course 4 in a previous semester, course 8 behaves as a free course, and the set of decisions to consider is $\{8, \bar{8}\}$. Thirdly, **Figure 4** represents the most complex relationship. As is shown, course 2 is a co-requisite for course 3, meaning it must be taken alongside course 3 if course 3 is taken and if course 2 has not been completed yet. From this relationship there are

three possible decisions to make, denoted in the sets $\{2, 3\}$, $\{2, \overline{3}\}$, and $\{\overline{2}, \overline{3}\}$. Either we take both courses, the first but not the other, or neither one of the courses.

*Geometry of Network*

Recalling prior research discoveries, the capability of representing decisions about courses as nodes in a network is a powerful concept. Because of the need to select a number of courses for the semester (arbitrarily set by user) and the types of relationships among courses, this required that the network have a certain geometry.

The first important dimension is the width, which is equal to the *budget B*, or the number courses the optimization tool selects per semester, plus one. This idea is shown in **Figure 5.**

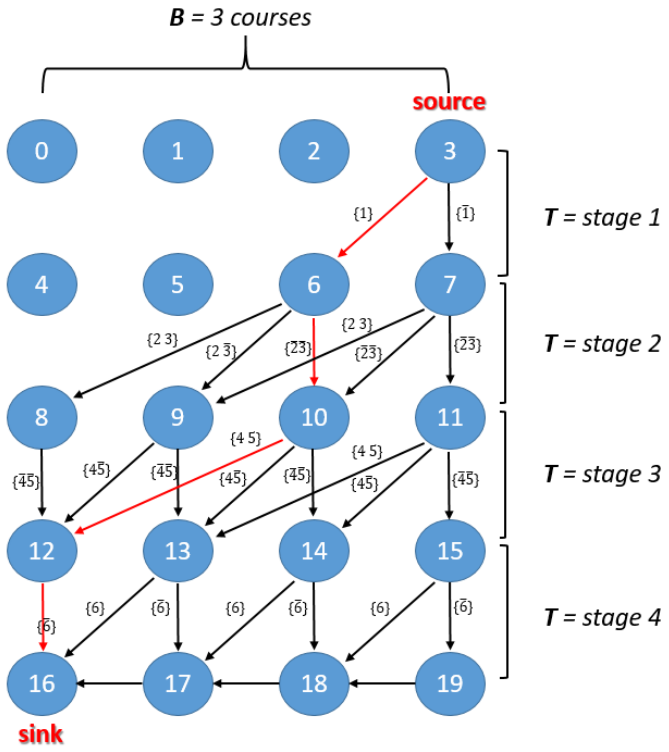


**Figure 5:** Conceptualization of the geometry of a network

Moving from one node to another such as along the red edge, definitive decisions to take a course result in advancing one column to the left, which corresponds to filling one course out of the budget allowance. Decisions to not take a course are shown as a vertical line pointing downward. This path does not correspond to filling one course out of the budget, so the path does not advance a column to the left. Edges that advance *two* columns to the left represent the definitive decision to take both courses in a co-requisite pair or relationship. In this case two positions out of the selection spaces are filled, resulting in advancing two columns to the left.

The other important dimension is the length of the network, also shown in **Figure 5**. This is equal to the number of stages (groups of alternatives, from which to select one) considered in the decision process plus one. Likewise in the preceding example, making the decision to take a course results in advancing one column to the left and also one row downward. Making the decision not to take a course also results in advancing one row downward. In either case, a decision was reached about the relationship considered. Over the course of any path from source to sink, the path advances over all levels, demonstrating the idea that all feasible course relationships are considered, and one and only one decision is reached.

*Generating the Network*

For the reason that the optimization model should be able to work for selecting any number of any courses, it was necessary to automate the construction of the network based on input course options and desired number of courses.

The approach taken was to input all course options, their relationships, and information such as course number, DFW rate, penalty, and whether or not the course had been completed, into a Course Log excel file. The main program, described in detail in the following section, read in the course information from the file to populate a three-dimensional array of all feasible alternative decisions for each relationship to be considered at each stage in the network.

Using all alternatives in the course options list, the program generated the $(B+1)(T+1)$ $x$ $(B+1)(T+1)$ adjacency matrix, which contained all the necessary information to build the network. Recalling, the adjacency matrix is a two dimensional matrix where an entry at the indices $ij$ represents an arrow pointing from node $i$ to node $j$ with cost $a_{ij}$ (zero indexing applies). Pertaining to an arc representing the decision for a single-course stage, the cost is comprised of the DFW rate $f$ of the course and the penalty $p$ for not taking it by the semester in question. Cost values for the lengths of edges in a pre-requisite or free course stage are calculated according to the following:

$$a_{ij} = cost = \begin{cases} fp & \rightarrow \text{course is taken} \\ p & \rightarrow \text{course is not taken} \end{cases}$$

The penalty values are not given in the data but are arbitrarily set by the user. In setting the penalty values for courses in a particular semester we effectively establish a class of course selection policies that are parameterized by these values. By exploring different parameter settings, we hope to identify course-selection policies that are efficient with respect to graduation metrics.

Likewise, the different edge lengths for arcs in a co-requisite stage (in which, say, course 1 is a co/requisite of course 2) are

$$a_{ij} = cost = \begin{cases} f_1 p_1 + f_2 p_2 \rightarrow both\ co-requisites\ taken \\ f_1 p_1 + p_2 \rightarrow first\ co-requisite\ taken \\ p_1 + p_2 \rightarrow neither\ co-requisite\ taken \end{cases}$$

From the preceding two equations we see that the arc length is the expected penalty due to a stage's courses either being attempted or unattempted but not completed after the current semester. The program then iterated through each entry of the adjacency matrix to build the network with the correct "arrows drawn" and path costs, stored in a Graph object of the Python class Graph. The network was then ready to run Dijkstra's algorithm to calculate the shortest path through the network.

As a visual example of network generation, a sample network representing the situation in which three courses are to be selected is shown in **Figure 6** below. In this situation, the selections will come from evaluating the stages of one free course, two co-requisite relationships, and a pre-requisite relationship.

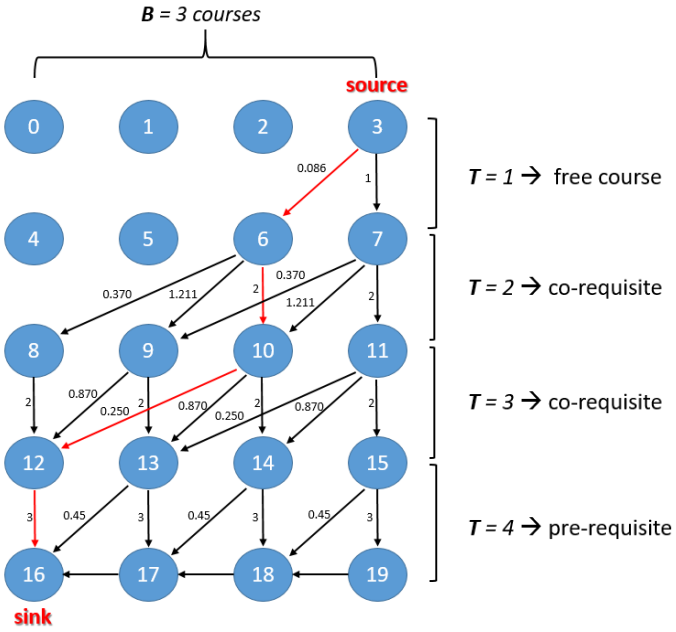


**Figure 6:** Sample network with costs

*Calculating the Shortest Path*

From **Figure 6** we can see that there are many ways to reach the sink from the source. Using the methodology from the preceding section, the program used Djikstra's algorithm to calculate the shortest path from the source to the sink.

The output of the algorithm is the optimal progression through the network. It is important to understand, however, that the path merely contains the numbers of the nodes of the optimal progression through the network. It does not contain the actual numbers of which courses to take. This information can be deduced by leveraging the geometry of the network.

*Extracting Which Courses to Take*

The fact that the network is constant in width allows for easy conversion from an optimal progression of nodes to optimal selection of courses to take. This idea is best explained by revisiting the prior example in **Figure 7.**

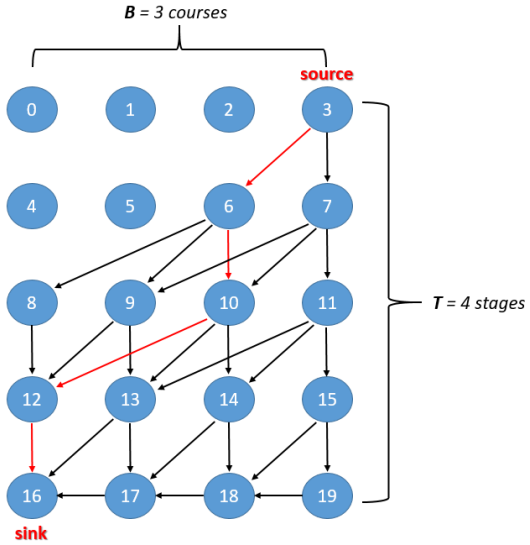


**Figure 7:** Optimal progression of nodes through network

Suppose that the shortest path, relating to the total of the costs of each segment, is the path in red. Also, suppose that in stages 1, 2, 3, and 4, the feasible alternatives were $\{1\}$ and $\{\bar{1}\}$ in stage 1, $\{2, 3\}$, $\{2, \bar{3}\}$, and $\{\bar{2}, \bar{3}\}$ in stage 2, $\{4, 5\}$, $\{4, \bar{5}\}$, and $\{\bar{4}, \bar{5}\}$ in stage 3, and $\{6\}$ and $\{\bar{6}\}$ in stage 4. This is an example of evaluating a free course, two co-requisite pair, and a pre-requisite pair. The shortest path as indicated is the path 3-6-10-12-16.

Here we notice that the difference between node numbers between successive nodes can be one of three cases: $B - 1$, $B$, or $B + 1$. The difference of $B$, such as between nodes 3 and 6 correspond to taking the first alternative, or taking course 1. The difference of $B + 1$, such as between nodes 6 and 10 correspond to not taking any courses evaluated in that stage; so neither course 2 nor 3 are taken. The third case, a difference of $B - 1$ where there are two courses in the alternative, such as between nodes 10 and 12, correspond to taking both courses 4 and 5.

*Simulating Progression through Course Subset*

Until this point, the methodology for how to return *one* semester's course decisions has been explained. However, in order to gather information about student success using the suggestions of the model, the optimization model had to iteratively model every semester until completion for a student. Thus, it was imperative to develop a method for the model to update its feasible alternatives as the student passes and/or fails courses.

This was accomplished in three major steps. First, to simulate a student proceeding through the course subset, random numbers were generated according to the pass distribution of each course in the output course selection. If the number was greater than the pass rate of that course, then the student passed that course. In this case, the program changed the 'Completed' attribute of the course in the Course Log to 1, for true.

Next, the courses that were completed had to be removed from the three-dimensional list that held the decision alternatives for each stage. This required a lot of intricate logic in the case where a student completed and passed only one of a co-requisite pair. In this case, all alternatives (3) were removed from the list for that stage, and the course in the co-requisite pair that was not taken was added back in the next stage.

Finally, three-dimensional list that stored the decision alternatives for each stage had to be updated with courses that the student was then able to take, such as courses that had pre-requisites that the student had just completed. To do this, the program revisited each course in the Course Log and added its alternatives to the list in a similar way as the first time.

This process of generating the feasible decision alternatives, calculating the shortest path, returning the courses to take, and updating the course decision alternatives with the new information was repeated until the student successfully completed all courses. In this way, a student's entire progression through the course subset with the recommendations of the model was modeled, and statistics were collected regarding how many semesters it took the student to "graduate."

*Verification of Model*

The model was validated by printing output from each critical step to the console. The critical steps to check were:

1. Populating the Course Options list, which stored the list of decision alternatives at every stage
2. Generating the adjacency matrix (contains information pertaining to which nodes point to which with which cost)
3. Generating the network
4. Calculating the shortest path through the network
5. Extracting which courses the node path corresponded to
6. Updating the Course Options list, after simulating and documenting pass/complete or fail/incomplete outcomes for courses
7. Adding course decision alternatives to Course Options that were not feasible the prior semester
8. Comparing “graduating rates” of different penalty assignment methods to select the best one

Each following step could not be completed until the preceding step was tried on many cases. More information pertaining to how each critical step was implemented in the program is in the following section.

# IMPLEMENTATION

## *Introduction*

This section outlines the implementation of the project methodology in the Python program. It summarizes key subroutines in the overall algorithm, discusses data structures used to implement methodologies, and explains the format of automation techniques.

## *Starter Code*

Due to the complexity of the algorithm and the fact that network implementation code is readily available on the internet, not all the code used in this project was developed by the researcher. The code that was used from the internet included a Python Graph class and associated methods, one of which was the key method Dijkstra. Below is a summary of the contents of the Graph class, and the full source code can be found in the Appendix:

- **__init__**(edges): the constructor for the Graph data type, which creates a Graph (network) with a total number of edges equal to *edges*
- **vertices**(): a property of a Graph object, which returns the number of total edges in the network
- **get_node_pairs**(n1, n2, both_ends=False): method, which returns a list of all node pairs between two node numbers for a directed network
- **remove_edge**(n1, n2, both_ends=False): method, which deletes an edge from a directed network
- **add_edge**(n1, n2, cost, both_ends=False): method, which adds an edge to a directed network
- **neighbours**(): Property, which returns sets of edges that each correspond to a group of edges
- **dijkstra**(source, dest): method, which returns a list of nodes corresponding to the shortest path from *source* to *dest* in a network

## *Original Code*

Below are the constructs and routines that were developed firsthand by the researcher for this project:

- **getSink**(b, t): method, which returns the number of the sink node of the network with given dimensions $(B+1)(T+1)$ x $(B+1)(T+1)$
- **CourseOptions**: a three-dimensional list of feasible course decisions in each stage
- **GenerateNetworkMatrix**(courseOptions, b, semNum): method, which returns the adjacency matrix to build a network considering all possible course decision outcomes in

*courseOptions*, which selects *b* courses, with cost values dependent upon the *semNum* th semester that the courses are taken

- Subroutine to generate the structure containing all feasible course decisions in each stage from an input list, *courseLog*
- Subroutine to build a Graph object from the adjacency matrix obtained from *GenerateNetworkMatrix*
- Subroutine to extract list of proposed courses from the returned node progression from *dijkstra*
- Subroutine to update *CourseOptions* by eliminating courses that have been passed/completed and adding in course decision alternatives that were not already feasible
- Subroutine to simulate students progressing through courses according to the suggestions of the model and calculate statistics

*Key Python Data Structures*

In the Python program, the construct *courseLog* is a structure that stores input data from an Excel file. It is of the type Dataframe, of the Pandas package. The Excel file records a list of all courses offered and information about each course. A sample of *courseLog* is shown in **Table 3.**

| CourseNum | RelType | CourseCode | PreReq | CoReq | CoReqFor | f | Semester1 | … | Semester7 | Completed |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | f | 2001 | | | | 0.09 | 1 | | 10 | 0 |
| 2 | c | 2103 | | | 3 | 0.21 | 1 | | 12 | 0 |
| 6 | p | 2333 | 3 | | | 0.11 | 0.05 | | 16 | 0 |

**Table 3:** Sample contents of *courseLog*, the input information

In this example, course 1 is a free course, as its relationship type is ‘f.’ This means it has no pre-requisites or co-requisites. It corresponds to the course INEG 2001. The DFW rate for this class is 9%. If not taken by semester it incurs a cost of 1, whereas if not taken by semester 7, it incurs a penalty of 10. The same logic is used for semesters 2-6, however the table is truncated for the sake of space. Likewise, information is stored for course 2, only that this course is a co-requisite for course 3. Similarly, the same applies to course 6, only that it has a co-requisite of course 3. The ‘Completed’ column records whether or not the course has been completed, where 0 represents false, and 1 represents true. The *courseLog* construct provides the information for generating the list of feasible course decision alternatives and updates information about each course as courses are completed.

In the program, the most important construct to understand is *CourseOptions.* This is the structure that stores all feasible course decision alternatives in each stage of determining a course selection for a semester. *CourseOptions* is of the datatype “list of lists of lists” of integers, which is most similar to a three-dimensional array of non-uniform length. The indices of each row of

*CourseOptions* store each decision alternative for a given stage, or a pre-requisite, co-requisite, or free course relationship. For example, below is an illustration of what the contents of *CourseOptions* could be:

*courseOptions* = [[[1], [-1]],
[[2, 3], [2, -3], [-2, -3]],
[[4, 5], [4, -5], [-4, -5]],
[[8], [-8]],
[[10], [-10]],
[[12], [-12]],
[[15], [-15]],
[[16], [-16]],
[[18], [-18]],
[[20], [-20]],
[[23], [-23]]]

Looking at the contents of *CourseOptions*, the first row represents a set (list) of all decision alternatives for stage one. The first alternative, denoted by the list [1] represents the alternative of taking course 1, as the value is positive. The second alternative, denoted by the list [-1] represents the alternative of not taking course 1. Stage two represents all alternatives from considering a co-requisite relationship of courses 2 and 3. The first alternative [2, 3] represents taking courses 2 and 3; the second alternative [2, -3] represents taking course 2 but not taking course 3; and the third alternative [-2, -3] represents neither taking course 2 nor course 3. The first and foremost goal of the program is to calculate exactly which single alternative should be selected from each row of *CourseOptions* in order to determine the optimal course selection.

The adjacency matrix is of the type two-dimensional list, or array. As stated before, for weighted networks of $N$ nodes, the network can be stored in an $N \times N$ matrix [2]. Each component $a_{ij}$ represents an edge from node $i$ to node $j$ [2]. The values in the adjacency matrix consist of either numbers or 'None' (null), where the number indicates the *cost* of the path from node $i$ to node $j$, and 'None' means no such connection exists. For example, the adjacency matrix for the first two stages of the complete network of all stages is the matrix below. The corresponding incomplete network is shown below:

column index
indicates node *j*

row index
indicates
node *i*

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | None | None | None | None | None | None | None | None | None | None | None | None |
| **1** | None | None | None | None | None | None | None | None | None | None | None | None |
| **2** | None | None | None | None | None | None | None | None | None | None | None | None |
| **3** | None | None | None | None | None | None | 0.068 | 1 | None | None | None | None |
| **4** | None | None | None | None | None | None | None | None | None | None | None | None |
| **5** | None | None | None | None | None | None | None | None | None | None | None | None |
| **6** | None | None | None | None | None | None | None | None | 0.370 | 0.211 | 2 | None |
| **7** | None | None | None | None | None | None | None | None | None | 0.370 | 0.211 | 2 |
| **8** | None | None | None | None | None | None | None | None | None | None | None | None |
| **9** | None | None | None | None | None | None | None | None | 0 | None | None | None |
| **10** | None | None | None | None | None | None | None | None | None | 0 | None | None |
| **11** | None | None | None | None | None | None | None | None | None | None | 0 | None |

In the adjacency matrix, the indices *ij* where the value is not 'None' represent an edge from node *i* to node *j* with a cost of $a_{ij}$. From the matrix, we see that $a_{36}$ signifies that node 3 points to node 6, and this matches the network shown in **Figure 8**. This means that the decision to take course 1 is represented by the path from node 3 to node 6 and incurs a cost of 0.086 (read in from *courseLog*). Likewise, we see that $a_{69}$ signifies that node 6 points to node 9 with a cost of 0.211. This matches the figure. This is the scenario of taking course 2 but not course 3.

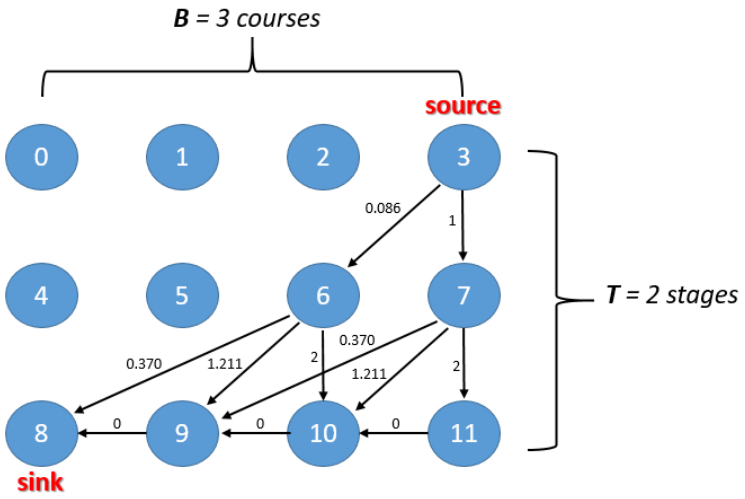


**Figure 8:** Network represented by preceding adjacency matrix

The network shown in **Figure 8** is of the datatype Graph, as defined in the Graph class. A graph, or network, is created when the constructor is called. Edges are added when the add_edge

function is called to the network object and the start and end node numbers are specified. In the program, the network is generated by looping through the adjacency matrix values $a_{ij}$ to add edges to the network. This is accomplished by the following call to the network object:

graph.add_edge(i, j, info_matrix[i][j], False)

The edge points from node *i* to node *j* and the cost of the path is equal to the adjacency matrix *info_matrix*[i][j]. The edge is directed.

A Python List, analogous to a one-dimensional array, is used in the program to store the output of the *dijkstra* function, which returns the optimal node number progression through the network. An additional list *coursesToTake* is also used to store the course selection, extracted from the optimal node progression list.

*Algorithm for Optimal Course Selection*

One pass through the network corresponds to one calculation of *B* optimal courses to take that semester. After understanding the data structures and methodologies in the preceding sections, below is the summary of the algorithm to arrive at the output list of optimal courses *C*.

$For\ each\ stage\ T\ in\ set\ of\ courses\ L$

$Append\ sequence\ of\ alternatives\ D\ to\ course\ options\ O$

$For\ all\ d \in D$:

$Construct\ and\ add\ node\ n\ \{n_1, n_2\}\ to\ set\ of\ all\ nodes\ N\ of\ network$

$Use\ Dijkstra's\ algorithm\ to\ return\ shortest\ path\ sequence\ S$

$For\ each\ s \in\ S_{n+1} - S_n$ :

$If\ s = B + 1 \rightarrow next\ s$

$If\ s = B \rightarrow add\ O_{T_{1_1}}\ to\ set\ of\ courses\ to\ take\ C$

$If\ s = B - 1 \rightarrow add\ O_{T_{1_1}}\ and\ O_{T_{1_2}}\ to\ C$

$Return\ C$

The algorithm loops through each line *L* in *CourseLog* to discover what stages *T* (relationships) exist among courses and adds the sequence *D* of all decision alternatives *d* to

*CourseOptions O* by stage. Next the algorithm constructs the corresponding edges $\{n_1, n_2\}$ of the network representing all possible decision paths. Dijkstra's algorithm returns the optimal node progression *S*. The courses to take are deduced from the node progression by determining which of three cases the difference between subsequent node numbers is. If the difference equals *B* or *B-1* then either add one or both course numbers from stage *T* of *O* to the set of courses to take *C*. Return *C*.

*Algorithm for Complete Progression Through Course Subset*

Progression through the entire course subset according to the suggestions of the model is achieved by simulating multiple semesters using the previous algorithm. The previous algorithm suggests a set *C* of courses to take each semester and the set of sequences *D* of decision alternatives *d* is updated as courses are completed and pre- and co-requisites become feasible. The algorithm is summarized below:

$For\ each\ T \in L$:

$Append\ sequence\ of\ alternatives\ D\ to\ course\ options\ O$

$While\ incomplete\ courses\ remain$:

$C = \ Algorithm\ For\ Optimal\ Course\ Selection$

$For\ c \in C$:

$num \sim N(0,1)$

$If\ num > f,\ then$

$L[completed] = true$

$For\ c\ \in C$:

$If\ c\ is\ completed, then$

$Remove\ d \in O\ where\ c \in d$

$numSemesters\ +\!= 1$

$Return\ numSemesters$

# RESULTS

## *Introduction*

This section is about the output of the model and how it can be used to draw insights.

## *Output from One Complete Run*

Below is sample output from running the program to completion. For each semester in simulation, the output contains the feasible course options for that student in that semester, the node number progression the student should take through the network, and *B* optimal courses that the optimal path corresponds to. In this particular run, *B* was set to 5 courses.

The 25 courses considered was a list based mostly on experimental data, with four extra courses that were based on similar courses, but not actual courses of the IE degree program. The failure rates and pre-req relationships for the real courses were populated by using the experimental data and the relationships in the current IE degree program, and the average failure rate was used for the courses that are not real. Information about these classes is detailed in the Data Collection and Preparation section. Penalty values were chosen largely based on pre-requisite chains and courses that absolutely had to be taken certain semesters, such as Applied Probability and Statistics for Engineers I, Simulation, Intro to Optimization, and IE Capstone Experiences I and II. Courses in pre-requisite chains and these courses were assigned high penalty values in the respective semesters in order to encourage the model to select these courses in the semesters they were essentially nonnegotiable. All other courses were assigned penalty values of 1, as they did not influence other courses and could be taken any time. The penalty values were assigned in a similar fashion to **Table 4:**

| Course | Sem1 | Sem2 | Sem3 | Sem4 | Sem5 | Sem6 | Sem7 | Sem8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 500 | 400 | 300 | 200 | 100 | 1 | 1 | 1 |
| 3 | 1 | 500 | 400 | 300 | 200 | 100 | 1 | 1 |
| 4 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 5 | 1 | 1 | 400 | 300 | 200 | 100 | 1 | 1 |
| 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 7 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 8 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 9 | 1 | 1 | 1 | 1 | 1 | 1 | 2000 | 1 |
| 10 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 3000 |

**Table 4:** Penalty values for instance run

The model was not run using any courses in co-requisite relationships for the reason that this is the only piece of logic that has yet to be implemented. At this time, it is difficult to assess the ability of the model to predict what a student should take when the courses involve co-requisites. However, the model fully implements the selection of free courses and pre-requisites, an instance of which is shown below.

For illustration, the output from the first three iterations of simulating students pass through 25 free and pre-requisite courses is shown below:

*courseOptions:*

*[[1], [-1]]*

*[[2], [-2]]*

*[[4], [-4]]*

*[[5], [-5]]*

*[[6], [-6]]*

*[[7], [-7]]*

*[[8], [-8]]*

*[[11], [-11]]*

*[[12], [-12]]*

*[[13], [-13]]*

*[[14], [-14]]*

*[[16], [-16]]*

*[[18], [-18]]*

*[[19], [-19]]*

*[[20], [-20]]*

*[[21], [-21]]*

*[[22], [-22]]*

*[[23], [-23]]*

*[[24], [-24]]*

*[[25], [-25]]*

*The student should take this path through the network:*

*[5, 11, 16, 21, 27, 33, 38, 44, 50, 55, 61, 66, 72, 78, 84, 90, 96, 102, 108, 114, 120]*

***This path corresponds to these courses:***

***[2, 4, 7, 12, 14]***

*courseOptions:*

*[[1], [-1]]*

*[[5], [-5]]*

*[[6], [-6]]*

*[[8], [-8]]*

*[[11], [-11]]*

*[[13], [-13]]*

*[[16], [-16]]*

*[[18], [-18]]*

*[[19], [-19]]*

*[[20], [-20]]*

*[[21], [-21]]*

*[[22], [-22]]*

*[[23], [-23]]*

*[[24], [-24]]*

*[[25], [-25]]*

*[[3], [-3]]*

*The student should take this path through the network:*

*[5, 11, 16, 21, 27, 33, 38, 44, 50, 55, 61, 66, 72, 78, 84, 90, 96]*

***This path corresponds to these courses:***

***[5, 6, 13, 19, 21]***

*courseOptions:*

*[[1], [-1]]*

*[[8], [-8]]*

*[[11], [-11]]*

*[[16], [-16]]*

*[[18], [-18]]*

*[[20], [-20]]*

*[[22], [-22]]*

*[[23], [-23]]*

*[[24], [-24]]*

*[[25], [-25]]*

*[[3], [-3]]*

*[[13], [-13]]* ← *student failed course 13 the previous semester, so it is an option.*

*[[15], [-15]]*

*[[17], [-17]]*

*The student should take this path through the network:*

*[5, 11, 16, 21, 27, 33, 38, 44, 50, 55, 61, 66, 72, 78, 84]*

***This path corresponds to these courses:***

***[8, 11, 20, 24, 3]***

*courseOptions:*

*[[1], [-1]]*

*[[16], [-16]]*

*[[18], [-18]]*

*[[22], [-22]]*

*[[23], [-23]]*

*[[25], [-25]]*

*[[13], [-13]]*

*[[15], [-15]]*

*[[17], [-17]]*

*[[3], [-3]]* ← *student failed course 3 the previous semester, so it is an option.*

...

Here we see that the student initially does not have credit for any courses, so their initial course options are all courses satisfying pre-requisite and/or free course requirements. For the first semester, the model recommends taking courses 2, 4, 7, 12, and 14. As the student completes these courses, the model updates *courseOptions* to reflect that courses 2, 4, 7, 12, and 14 are no longer options to take and have been marked as completed in *CourseLog*. During simulation, the model detects that the student failed courses 13 and 3 in semesters 2 and 3 respectively, and therefore does not mark them complete. Because of this, courses 13 and 3 show up again in the next semesters' *courseOptions*. The same process is iterated until all courses are passed and completed.

After simulating 100 students, the average number of semesters it took students to proceed through all courses was 5.9 semesters. The results are summarized in **Figure 9**:

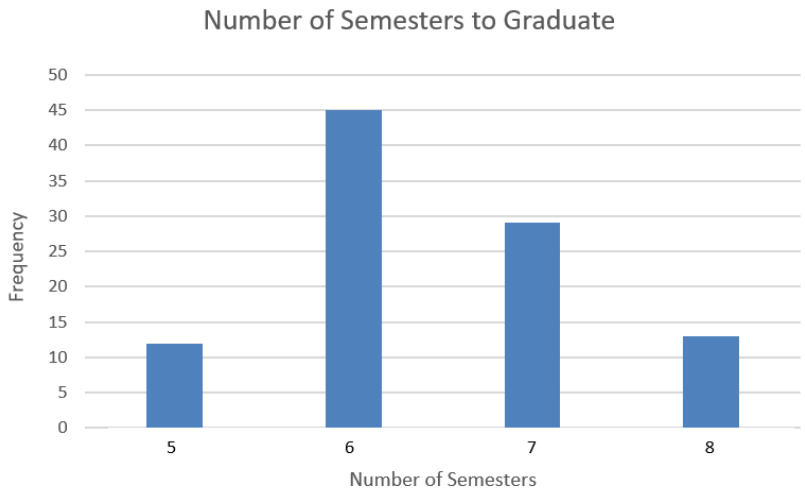


**Figure 9:** Frequency chart for number of semesters to graduate

It is not surprising to see that the mean number of semesters to graduate was 5.9 when the total number of courses to take was 25. If all courses were passed on the first time, the minimum number of semesters to graduate would have been 5. Since all courses have a non-zero DFW rate, it would not be expected for the mean number of semesters to graduate across 100 students to be 5. From the data we can conclude that using the suggestions of the model, 86% of the students graduated within an extra semester, and 100% graduated within an extra year.

This suggests that the model can provide good insight as to what a student should take in order to have the best probability of graduating on time. This number is not far from reality, with most students graduating within an extra year of the eight-semester plan if they do not graduate on time. Perhaps another reason why the mean number of semesters to graduate is higher than expected is because the model does not account for students being able to “make up time” by taking courses over summer break, winter, or taking extra hours. For the model to be able to produce results that are reasonable without accounting for students making up for failed courses, the model is deemed a success and can only be improved.

## MODEL ASSUMPTIONS AND CONCLUDING REMARKS

### *Introduction*

This section describes three important limitations of the optimization model and areas for improvement, using this study as a baseline.

### *Assumption*

To maintain simplicity of the baseline, this model is based on the assumption that all courses are offered every semester. However, this is not the case in reality. This should be taken into consideration when using this model to plan semester courses. The solution to alleviating this assumption would be to include in the *Course Log* binary variables for each course, indicating whether or not they are offered in each semester. This way, logic would prevent these course decision alternatives from being added to *Course Options* in semesters they are not offered.

### *Assumption*

This model also assumes that all courses in the input list must be taken and that no courses are electives. The model works off this assumption for the reason that data was unavailable on non-industrial engineering courses, but in reality, the industrial engineering degree contains non-industrial engineering courses. A simple way for the model to work without relying on this assumption would be to implement logic forcing the model to select courses that are on the required list and a certain number from an optional list. This ensures that the number of course selections for the model to output is less than the number of courses available, showing that the model can handle both required and elective courses.

### *Assumption*

This model is limited to only selecting industrial engineering courses, however, in reality an industrial engineering degree contains non-industrial engineering courses. This was done as data was unavailable for non-industrial engineering courses. The solution to alleviating this assumption is to also include courses that are non-industrial engineering. This way it is more representative of a real industrial engineering degree.

### *Concluding Remarks*

Although the model is simplified and only represents progression through a subset of courses, it still abides by the vast majority of logic in a real life scenario. The model

accomplishes its goal of optimizing and suggesting the courses a student should take in order to have the best chance of passing all courses and staying on track. Upon further development, the model could be improved by incorporating non-industrial engineering courses, enforcing a minimum number of required and the rest elective courses, and restricting which courses are available each semester. The results of the complete model could then be fully validated against the standard eight semester plan. Until then, however, the current developed model accomplishes the toughest modeling requirements and can be improved to solve the remaining gaps in logic with minimal effort.

APPENDIX

*Introduction*

This section contains two sections, one for the code that was not written by the researcher, and original code that was written by the researcher. The code that was not original included the Python Graph class and a separate implementation of Dijkstra's algorithm. The code written by the researcher included the main program that read information from a user and output the progression the student should take through the sample course subset.

*Starter Code*

```
def make_edge(start, end, cost=1):
    return Edge(start, end, cost)


def getSink(b, t):
    return ((b+1)*(t+1)) - 1 - b


class Graph:
    def __init__(self, edges):
        # let's check that the data is right
        wrong_edges = [i for i in edges if len(i) not in [2, 3]]
        if wrong_edges:
            raise ValueError('Wrong edges data: {}'.format(wrong_edges))


        self.edges = [make_edge(*edge) for edge in edges]


    @property
    def vertices(self):
        return set(
            sum(
                ([edge.start, edge.end] for edge in self.edges), []
            )
```

```
    )

def get_node_pairs(self, n1, n2, both_ends=False):
    if both_ends:
        node_pairs = [[n1, n2], [n2, n1]]
    else:
        node_pairs = [[n1, n2]]
    return node_pairs

def remove_edge(self, n1, n2, both_ends=False):
    node_pairs = self.get_node_pairs(n1, n2, both_ends)
    edges = self.edges[:]
    for edge in edges:
        if [edge.start, edge.end] in node_pairs:
            self.edges.remove(edge)

def add_edge(self, n1, n2, cost, both_ends=False):
    node_pairs = self.get_node_pairs(n1, n2, both_ends)
    for edge in self.edges:
        if [edge.start, edge.end] in node_pairs:
            return ValueError('Edge {} {} already exists'.format(n1, n2))

    self.edges.append(Edge(start=n1, end=n2, cost=cost))
    if both_ends:
        self.edges.append(Edge(start=n2, end=n1, cost=cost))

@property
def neighbours(self):
    neighbours = {vertex: set() for vertex in self.vertices}
    for edge in self.edges:
        neighbours[edge.start].add((edge.end, edge.cost))
```

```
    return neighbours


def dijkstra(self, source, dest):
    assert source in self.vertices, 'Such source node doesn\'t exist'
    distances = {vertex: inf for vertex in self.vertices}
    previous_vertices = {
        vertex: None for vertex in self.vertices
    }
    distances[source] = 0
    vertices = self.vertices.copy()

    while vertices:
        current_vertex = min(
            vertices, key=lambda vertex: distances[vertex])
        vertices.remove(current_vertex)
        if distances[current_vertex] == inf:
            break
        for neighbour, cost in self.neighbours[current_vertex]:
            alternative_route = distances[current_vertex] + cost
            if alternative_route < distances[neighbour]:
                distances[neighbour] = alternative_route
                previous_vertices[neighbour] = current_vertex

    path, current_vertex = deque(), dest
    while previous_vertices[current_vertex] is not None:
        path.appendleft(current_vertex)
        current_vertex = previous_vertices[current_vertex]
    if path:
        path.appendleft(current_vertex)

    return path
```

*Original Code:*

```
        def GenerateNetworkMatrix(self, courseOptions, b, semNum):
    CN = b
    t = len(courseOptions)

    courseLog = pandas.read_excel('CourseLookupSecond.xlsx')
    #penaltyValues = pandas.read_excel('PenaltyValuesArbitrary.xlsx')

    binaryVals = []
    for i in range(0, len(courseOptions)):
       if len(courseOptions[i]) == 3:
          binaryVals.append(1)
       else:
          binaryVals.append(0)

    # Creates a bxb array initialized to Null ("None")
    matrix = [[None for x in range((b+1)*(t+1))] for y in range((b+1)*(t+1))]          #changed this 3/11

    for i in range(0, t):                                    #changed b to t
       #populating network with arrows
       while CN <= (b*i + b + i) and CN <= (getSink(b, t) + b):     #make sure CN is less than the end node # and less than the very last node
          if binaryVals[i] == 0:     #check what kind of relationship in this stage--> 0 for free/pre-req, 1 for co-req
             if (CN+b > b*i+b+i) and (CN+b+1 > b*i+b+i):     #if both arrows go to next line/anywhere else but far left
                matrix[CN][CN+b] = courseLog.iloc[courseOptions[i][0][0] - 1]['f'] * courseLog.iloc[courseOptions[i][0][0] - 1]['Semester' + str(semNum)]     #lookup value
                matrix[CN][CN+b+1] = courseLog.iloc[courseOptions[i][0][0] - 1]['Semester' + str(semNum)]
             else:     #at the far left node on each row
                #if (CN+b > b*i+b+i):
```

```
                #matrix[CN][CN+b] = courseLog.iloc[courseOptions[i][0][0] - 1]['f']

            if (CN+b+1 > b*i+b+i):

                matrix[CN][CN+b+1] = courseLog.iloc[courseOptions[i][0][0] - 1]['Semester' + str(semNum)]


        CN = CN + 1

    elif binaryVals[i] == 1:

        if (CN+b-1 > b*i+b+i) and (CN+b > b*i+b+i) and (CN+b+1 > b*i+b+i):

            matrix[CN][CN+b-1] =
(courseLog.iloc[courseOptions[i][0][0]]['f']*courseLog.iloc[courseOptions[i][0][0]]['Semester' + str(semNum)]) +
(courseLog.iloc[courseOptions[i][0][1] - 1]['f']*courseLog.iloc[courseOptions[i][0][1] - 1]['Semester' +
str(semNum)])

            matrix[CN][CN+b] = courseLog.iloc[courseOptions[i][0][0] -
1]['f']*courseLog.iloc[courseOptions[i][0][0] - 1]['Semester' + str(semNum)] +
courseLog.iloc[courseOptions[i][0][1] - 1]['Semester' + str(semNum)]

            matrix[CN][CN+b+1] = courseLog.iloc[courseOptions[i][0][0]]['Semester' + str(semNum)] +
courseLog.iloc[courseOptions[i][0][1] - 1]['Semester' + str(semNum)]

        #if (CN+b-1 > b*i+b+i):

            #matrix[CN][CN+b-1] = courseLog.iloc[courseOptions[i][0][0] - 1]['f'] +
courseLog.iloc[courseOptions[i][0][1] - 1]['f']

        elif (CN+b > b*i+b+i) and (CN+b+1 > b*i+b+i):

            matrix[CN][CN+b] = (courseLog.iloc[courseOptions[i][0][0] -
1]['f']*courseLog.iloc[courseOptions[i][0][0] - 1]['Semester' + str(semNum)]) +
courseLog.iloc[courseOptions[i][0][1] - 1]['Semester' + str(semNum)]

            matrix[CN][CN+b+1] = courseLog.iloc[courseOptions[i][0][0]]['Semester' + str(semNum)] +
courseLog.iloc[courseOptions[i][0][1] - 1]['Semester' + str(semNum)]


        elif (CN+b+1 > b*i+b+i):

            matrix[CN][CN+b+1] = courseLog.iloc[courseOptions[i][0][0]]['Semester' + str(semNum)] +
courseLog.iloc[courseOptions[i][0][1] - 1]['Semester' + str(semNum)]

        else:

            pass



        CN = CN + 1


    else:
```

```
                CN = CN + 1

        #add backwards arrows prohibiting going backwards



        #add arrows on bottom pointing to sink
        for i in range(0, b):
            matrix[getSink(b, t)+b-i][getSink(b, t)+b-i-1] = 1

        return matrix




#---------------------------------MAIN-------------------------------------------------------------------

    #for reference, the structure of courseOptions
    '''courseOptions = [[[1], [-1]],
                [[2, 3], [2, -3], [-2, -3]],
                [[4, 5], [4, -5], [-4, -5]],
                [[8], [-8]],
                [[10], [-10]],
                [[12], [-12]],
                [[15], [-15]],
                [[16], [-16]],
                [[18], [-18]],
                [[20], [-20]],
                [[23], [-23]]]'''



budget = 5
```

```
graph = Graph(())
courseLog = pandas.read_excel('CourseLookupSecond.xlsx')
courseOptions = []
numSemesters = 1




    #if we don't have 5 courses left to take, need to build network of different dimension
    #if len(courseLog[courseLog['Completed'] == 0]) < 5:
        #budget = len(courseLog[courseLog['Completed'] == 0])



    #form course options
for i in range(0, len(courseLog.index)):
    #co reqs
    if courseLog.iloc[i]['RelType'] == 'c' and courseLog.iloc[i]['Completed'] == 0:
        courseOptions.append([[int(courseLog.iloc[i]['CourseNum']), int(courseLog.iloc[i]['CoReqFor'])],
[int(courseLog.iloc[i]['CourseNum']), -1 * int(courseLog.iloc[i]['CoReqFor'])], [-1 *
int(courseLog.iloc[i]['CourseNum']), -1 * int(courseLog.iloc[i]['CoReqFor'])]])


    #pre-reqs
    elif courseLog.iloc[i]['RelType'] == 'p' and courseLog.iloc[int(courseLog.iloc[i]['PreReq']) - 1]['Completed'] == 1
and courseLog.iloc[i]['Completed'] == 0:
        courseOptions.append([[int(courseLog.iloc[i]['CourseNum'])], [-1 * int(courseLog.iloc[i]['CourseNum'])]])


    #free courses
    elif courseLog.iloc[i]['RelType'] == 'f' and courseLog.iloc[i]['Completed'] == 0:
        courseOptions.append([[int(courseLog.iloc[i]['CourseNum'])], [-1 * int(courseLog.iloc[i]['CourseNum'])]])
    else:
        pass
```

```
#loop while there are incomplete courses
while len(courseLog[courseLog['Completed'] == 0]) > 0: ###changed tonight

    #if we don't have 5 courses left to take, need to build network of different dimension
    if len(courseLog[courseLog['Completed'] == 0]) < budget: ###changed tonight
        budget = len(courseLog[courseLog['Completed'] == 0]) ###changed tonight

    print("courseOptions:")
    for i in courseOptions:
        print(i)
    print("")

    options = len(courseOptions)

    #matrix contains all arrow information
    info_matrix = graph.GenerateNetworkMatrix(courseOptions, budget, numSemesters)

    #generate actual network
    for i in range(0, len(info_matrix)):
        for j in range(0, len(info_matrix)):
            if info_matrix[i][j] != None:
                graph.add_edge(i, j, info_matrix[i][j], False)

    #return shortest node path
    nodePath = graph.dijkstra(budget, getSink(budget, options))     #this is a deque
    print("The student should take this path through the network:")
    print(list(nodePath))
    print("")

    #extract course numbers of what to take from node list
    coursesToTake = list()
    stage = 0
```

```
for i in range(0, len(nodePath) - 1):
    if nodePath[i+1] - nodePath[i] == budget + 1:
        stage = stage + 1
    elif nodePath[i+1] - nodePath[i] == budget:
        coursesToTake.append(courseOptions[stage][0][0])
        stage = stage + 1
    elif nodePath[i+1] - nodePath[i] == budget - 1:
        if len(courseOptions[stage][0]) > 0:
            coursesToTake.append(courseOptions[stage][0][0])

        if len(courseOptions[stage][0]) > 1:
            coursesToTake.append(courseOptions[stage][0][1])
        stage = stage + 1
    else:
        pass

print("This path corresponds to these courses:")
print(coursesToTake)

#update courseOptions

#first, update courseLog to reflect courses taken
for i in range(0, len(coursesToTake)):
    #generate random number to see if they passed
    num = random()

    #only mark completed if passed class
    if num > float(courseLog.iloc[coursesToTake[i] - 1]['f']):
        print(num)
        courseLog.set_value(int(coursesToTake[i] - 1), 'Completed', 1)
numSemesters = numSemesters + 1
```

```
    print("")
    print("")
    #reconfigure courseOptions to reflect courses that were taken

    #delete courses that have been taken
    for i in range(0, len(coursesToTake)):
        #check if adjacent courses are a co-req pair
        #check if at the end of the list
        if len(coursesToTake) - i != 1:
            if courseLog.iloc[int(coursesToTake[i]) - 1]['CoReqFor'] == int(coursesToTake[i+1]):
                for obj in courseOptions:
                    if obj[0][0] == coursesToTake[i] and obj[0][1] == coursesToTake[i+1]:
                        courseOptions.remove(obj)
            else:
                for obj in courseOptions:
                    if obj[0][0] == coursesToTake[i]:
                        courseOptions.remove(obj)
        else:
            for obj in courseOptions:
                if obj[0][0] == coursesToTake[i]:
                    courseOptions.remove(obj)



    print("")



    #add in new courses that student is now able to take --> check for co-reqs and pre-reqs
    for i in range(0, len(courseLog.index)):


        count = 0
        #test if the course is currently in courseOptions by counting # of occurrences. If it's not and pre-req has been
completed, add.
```

```
    for list_obj in courseOptions:
        for obj in list_obj:
            if int(courseLog.iloc[i]['CourseNum']) in obj:
                count = count + 1


    if count == 0:


        #3 possibilities for co-req relationships: passed both, passed #1 not #2, passed #2 not #1
        if courseLog.iloc[i]['RelType'] == 'c':
            if courseLog.iloc[i]['Completed'] == 1 and courseLog.iloc[int(courseLog.iloc[i]['CoReqFor']) -
1]['Completed'] == 1:
                pass
            elif courseLog.iloc[i]['Completed'] == 0 and courseLog.iloc[int(courseLog.iloc[i]['CoReqFor']) -
1]['Completed'] == 1:
                courseOptions.append([[int(courseLog.iloc[i]['CourseNum'])], [-1 *
int(courseLog.iloc[i]['CourseNum'])]])
            elif courseLog.iloc[i]['Completed'] == 1 and courseLog.iloc[int(courseLog.iloc[i]['CoReqFor']) -
1]['Completed'] == 0:
                courseOptions.append([[int(courseLog.iloc[i]['CoReqFor'])], [-1 * int(courseLog.iloc[i]['CoReqFor'])]])
            elif courseLog.iloc[i]['Completed'] == 0 and courseLog.iloc[int(courseLog.iloc[i]['CoReqFor']) -
1]['Completed'] == 0:
                courseOptions.append([[int(courseLog.iloc[i]['CourseNum']), int(courseLog.iloc[i]['CoReqFor'])],
[int(courseLog.iloc[i]['CourseNum']), -1 * int(courseLog.iloc[i]['CoReqFor'])], [-1 *
int(courseLog.iloc[i]['CourseNum']), -1 * int(courseLog.iloc[i]['CoReqFor'])]])
            else:
                pass
        #pre-reqs
        elif courseLog.iloc[i]['Completed'] == 0 and courseLog.iloc[i]['RelType'] == 'p' and
courseLog.iloc[int(courseLog.iloc[i]['PreReq']) - 1]['Completed'] == 1:
            courseOptions.append([[int(courseLog.iloc[i]['CourseNum'])], [-1 *
int(courseLog.iloc[i]['CourseNum'])]])


        #free courses
        elif courseLog.iloc[i]['RelType'] == 'f' and courseLog.iloc[i]['Completed'] == 0:
            courseOptions.append([[int(courseLog.iloc[i]['CourseNum'])], [-1 *
int(courseLog.iloc[i]['CourseNum'])]])
```

```
        else:
            pass



print("Number of semesters to graduate:  ", numSemesters - 1)
```

WORKS CITED

[1] ABİDİN, D., & ÇAKIR, H. Ş. (2014). Analysis of a rule-based curriculum plan optimization system with spearman rank correlation. Turkish Journal of Electrical Engineering & Computer Sciences, 22, 176-190. doi:10.3906/elk-1204-14

[2] ARORA, N., SINGH, N., Freelancer Trainer, Gurugram, Haryana, India, & Associate Professor, Department of Applied Sciences, World College of Technology and Management, Gurugram, Haryana, India. (2017).

[3] Boldyreva, Maria. "Dijkstra's Algorithm in Python: Algorithms for Beginners." DEV Community. DEV Community, July 11, 2018.

[4] Factors affecting the academic performance of college students. I-Manager's Journal of Educational Technology, 14(1), 47. doi:10.26634/jet.14.1.13586

[5] Belfield, C. R. (2007) The Price We Pay: Economic and Social Consequences of Inadequate Education. Brookings Institution Press.

[6] Farahani, Reza Zanjirani, and Elnaz Miandoabchi. *Graph Theory for Operations Research and Management Applications in Industrial Engineering*. Business Science Reference, 2013.

[7] Heckler, M. A. (2018). The importance of a college education.

[8] Hester, B. T., & Ishitani, T. T. (2018). Institutional expenditures and state economic factors influencing 2012–2014 public university graduation rates. Planning for Higher Education, 46(4), 41-47.

[9] Li, D., & Lu, M. (2017). Automated generation of work breakdown structure and project network model for earthworks project planning: A flow network-based optimization approach. Journal of Construction Engineering and Management, 143(1), 4016086. doi:10.1061/(ASCE)CO.1943-7862.0001214 8. Powell, B. A., D. S. Gilleland, and L. C. Pearson. 2012. Expenditures, Efficiency, and Effectiveness in U.S. Undergraduate Higher Education: A National Benchmark Model. Journal of Higher Education 83 (1): 102-127.

[10] Nykamp, Duane Q. Network definition – Math Insight: Math Insight.

[11] Pascarella, E. T., and P. T. Terenzini. 1991. How College Affects Students: Findings and Insights from Twenty Years of Research. San Francisco: Jossey-Bass.

[12] Planck, Max (Photograph). (2018). *Example of a directed network.*NetworkAnalyzer Online Help. https://med.bioinf.mpi-inf.mpg.de/netanalyzer/help/2.5/.

[13] The Increasing Importance of College Education. (2015, Dec 10). Mint Retrieved from http://0-search.proquest.com.library.uark.edu/docview/1747193173?accountid=8361

[14] Willcox, K. E., & Huang, L. (2017). Network models for mapping educational data. Design Science, 3 doi:10.1017/dsj.2017.18

[15] Yan, M. (2020). Dijkstra's Algorithm. Math.mit.edu.