Capstone Project – CSC 599 Final Report

Automated Academic Advisor

Yazid Boufous 202105562

Hiba Akbi 202004281

A report submitted in partial fulfillment of the degree of

BS in Computer Science

Supervisor: Dr. Khalil Mershed

Department of Computer Science and Mathematics

School of Arts and Sciences, Lebanese American University

May 17th , 2024

Declaration

We, Yazid Boufous and Hiba Akbi; hereby declare that

- This report has been prepared on the basis of our own work. All other published and unpublished source materials that have been used in this report were referenced or acknowledged.
- The work in this project was carried out in accordance with the regulations stated in the Lebanese American University student code of conduct.

Date of Submission: May 17th, 2024.

Signature: Hiba Akbi 

Yazid Boufous 

Dedication

We would like to first extend our utmost gratitude and appreciation to our capstone advisor, Dr. Khaleel Mershad, for his invaluable guidance and support throughout the development of this project. His expertise and insight were indispensable in overcoming the many challenges we encountered.

We appreciate that you believed in us and the potential of our project. When we initially proposed the idea and we saw how challenging it was, we had our doubts especially because it was not exactly how we envisioned it to be. However, your trust in our abilities, encouraging words, and the continuous support and resources you provided us in every meeting we had were pivotal in pushing the project toward success. We are extremely thankful for your mentorship and are honored to have had the opportunity to learn under your guidance.

Table of Contents

Abstract

Throughout their academic journey, students frequently encounter situations where they are required to make adjustments to their academic course projection plans. These modifications may arise due to factors such as unavailability of courses, repetition of courses, change of majors, addition of a minor, semester overload,... etc. Such changes are often pivotal for students as they lead to a change in their academic course plan, thus possibly delaying their graduation for additional semesters. The current common approach to managing academic projection plans is complicated, time-consuming, and highly prone to error as it is done manually, either by the students themselves or with the help of an advisor. The complexity of this manual process has shown an obvious need for an automated academic advising system. Such a system would not only facilitate the course projection planning process but also enhance its accuracy. By automating this process, advisors would be less pressured, allowing them to focus on more complex advising needs, thus improving the overall quality of academic support services.

Our Automated Academic Advisor comprehends a student's academic background, including majors, completed credits, and the anticipated graduation timeline, integrating this with an understanding of major requirements, course prerequisites, co-requisites, and the different rules and guidelines set by the university. In addition, it caters to student input to customize their course projection plan regarding credit load and summer preferences. The primary limitation of the system is that it is tailored to the Undergraduate Computer Science Degree at LAU, which does not fully represent the entire university's catalog. However, the aim is to create a generalized framework that could later be adapted to different majors, minors, and requirements. Initial testing shows that the system effectively automates the process of course planning, minimizing the risk of human error and enhancing the accuracy of these plans. This tool is a valuable tool for educational institutions looking to enhance their academic advising processes.

Keywords: course planning, academic advising, graduation plan, graph-based modeling

1. Introduction

Efficient and accurate academic course planning is important not only for students but also for universities. For students, having a well-organized course plan can lead to increased satisfaction with their educational experience, improving graduation rates, and reducing the time to degree completion. For universities, effective academic advising can enhance student retention. However, this process is complicated and lengthy, because, when students usually try to customize and modify their projection plans, they are not always aware of the rules and constraints that exist between the different courses and that are set by the university, and often need the help of their advisors. Advisors assess all the courses and modify the course plan by doing the proper calculations while abiding by the rules and restrictions such as prerequisites, credit limit, and graduation date. This manual process places a significant burden on advisors, particularly during peak advising periods such as the beginning of registration for students. The consequences of errors in this process are substantial, leading to a delay in a student's progress toward graduation and financial and personal strain.

2. Representation of the Problem

2.1. Modeling the problem as a Graph Problem

After doing a literature review of the project idea to see any previously published papers or already implemented and existing platforms, we came across various implementations that address the same topic. However, the most common approach that was used was using Linear Programming, an approach that uses linear equations to determine how to arrive at the optimal situation as an answer to a mathematical problem, and it was represented as a mathematical issue.

In our current approach that seemed more intuitive, we employ graph theory to represent course prerequisites and corequisites, ensuring all academic requirements are met without violations. Utilizing heuristics such as limiting

credits per semester and prioritizing courses based on their reachability within the graph traversal, the system adapts to specific student requests, like semester-specific course inclusion or summer preferences. The algorithms are designed to handle complex requirements and multiple constraints efficiently, providing optimal solutions that consider academic rigor and personal student preferences.

2.2. Representation of Courses and Course Requirements

Courses as Nodes

In our graph representation, a course is represented by a node. Course details such as the ID of the course, the name of the course, its number of credits, and other course details such as the list of prerequisites and corequisites are represented by fields in the Course.java class.

For the sake of clarity and reusability, we draw a distinction in our code between a Course and a Node, by representing them by two different Classes (Course.java and Node.java). The course object is meant to represent a course in our database, independently of how it interacts with the graph structure or the projection plan. Think of it as the simple representation of a course in LAU's Course Catalog. On the other hand, a Node object is a more complex object whose field values are related to the graph structure and projection plan; in other words, a Node doesn't exist independently of the graph structure. A node object includes attributes such as a 'course' (of type Course) which is the course this node is representing in the graph, a 'level' representing the level of the course in the level map, a 'reachability' which represents the number of course this node is a prerequisite for whether directly or indirectly, and a list of Edges to represent co-requisites and prerequisites.

For now, we are not interested in explaining in detail the meaning and explanation behind the computation of each field value, as we will expand on this extensively in the upcoming sections. We will limit this section to explaining how Courses and course requirements such as prerequisites and corequisite exist in our graph. For this purpose, and within most of this report, we will be using the terms 'Course' and 'Node' interchangeably.

Prerequisites and Corequisites as Weighted Edges

Our graph structure G(V,E) includes a list of vertices V and a list of Edges E. A vertex is a Node, which as explained earlier represents a course along some other fields that help us process and traverse the graph as needed. Edges on the other hand represent the relationship between our courses. We have two types of relationship between courses: a prerequisite relationship, and a co-requisite relationship.

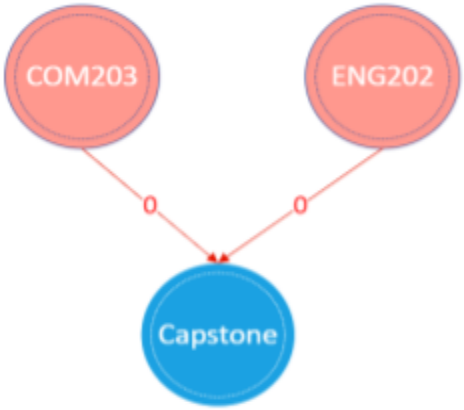
Prerequisite Relationship: A course A is called a prerequisite to Course B if B cannot be taken in a semester (i) unless A has already been completed in a previous semester (j) such that j<i.

In our graph, the prerequisite relationship is represented as an outgoing edge from A to B with a weight of 1: A → B, see figure.

Representation of the prerequisite relationship
Discrete Structures 1 is a prerequisite to Discrete Structures 2, it is represented with a an outgoing edge of weight 1.

Corequisite Relationship: A course A is called a corequisite to Course B if B cannot be taken in a semester (i) unless A has already been completed in previous semesters (such that j<i) OR A is also taken in the same semester (i). To simplify, a course A is called a prerequisite to Course B if B cannot be taken in a semester (i) unless A has already been completed in a semester (j) such that j≤i.

In our graph, the prerequisite relationship is represented as an outgoing edge from A to B with a weight of 0: A → B, see figure.

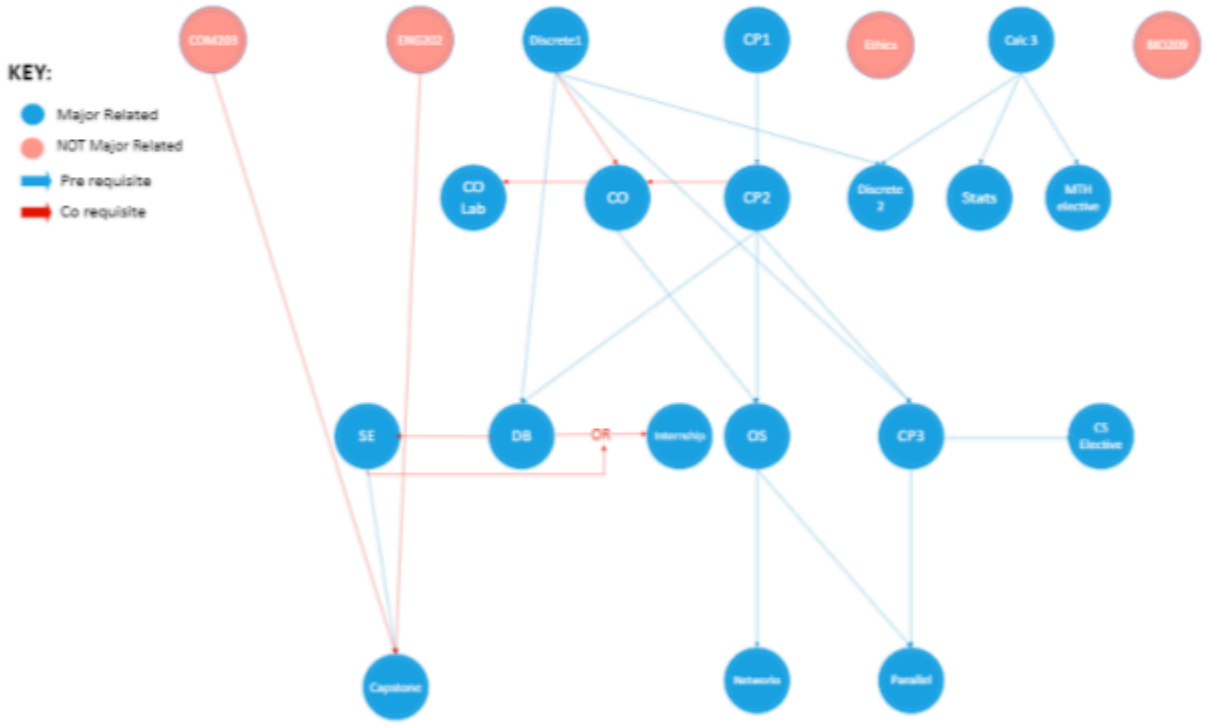


Representation of the corequisite relationship
ENG 202 and COM 203 are corequisites of the Capstone Course, they are represented with an outgoing edge of weight 0.

It is worth noting that in the corequisite relationship (and trivially in the prerequisite relationship), A→B doesn't imply that B→A. For example, the CSC245: Objects & Data Abstraction (commonly referred to as CP2) is a corequisite to the CSC320: Computer Organization Course (commonly referred to as CO).

This relationship means that CO cannot be taken in a semester unless CP2 is being currently taken at the same time, or CP2 has already been taken in previous semesters. The inverse however does not hold, as CP2 can be taken at any point in the semester without catering for CO.

Overall, the representation of the Computer Science Degree is represented in the graph below. We note that for the sake of our project, we relied on the old curriculum, which details can be found on the LAU Website.



Graph Representation of the Computer Science Curriculum Requirements

2.3. Projection Plan Heuristics

What are Heuristics

When creating a projection plan, it is necessary to consider a number of restrictions. These include ‘Course Restrictions’ like corequisite and prerequisite relationships which ensure that courses are taken in the correct order, but also other restrictions that concern the institution’s academic regulations, which we will refer to as ‘LAU Restrictions’. Some examples would be the maximum credits limit in a semester or the minimum number of semesters to graduation.

Both Course Restrictions and LAU Restrictions are mandatory to handle, as a student projection plan cannot bypass them. However, there exist other types of rules which are not mandatory but would be preferable to consider. For example, registering for a semester with a combination of major courses, minor courses, and LAC courses is preferable to registering for a

semester with a full load of major courses, as it allows the student to distribute the load of courses in a balanced manner which will facilitate their experience and likely lead to better academic performance. This restriction, however, is not mandatory and should be discarded if it prevents the student from graduating on time.

Hence, we draw a distinction between these two types of heuristics and define them as follows:

Mandatory Heuristics: Include Course Restrictions as well as LAU Restrictions. Cannot be bypassed.

Optional Heuristic: Include all other restrictions that help improve the projection plan and semester balancing. Can be discarded as needed.

In the section below, we will list and explain in detail the heuristics that have been applied in our system, then explain their code implementation in Section 3.3 Iterative search for a solution and Heuristic Relaxation.

Mandatory Heuristics (M.H)

As explained above, mandatory heuristics are all of the rules and regulations that must, without fail, be handled when creating a projection plan. They include restrictions originating from the courses themselves, like their prerequisite and corequisite relationship, but also other rules that we have extracted from LAU's policies. If a projection plan cannot be generated within the given graduation deadline without transgressing one of these heuristics, then the graduation deadline must be adjourned an extra semester.

M.H.1 Prerequisite and Corequisite

Heuristic Description: *"Courses cannot transgress the order defined by their prerequisite and corequisite relationships"*

Explanation: This of course is logical, as a projection plan is only valid if courses are taken in the correct order.

M.H.2 Labs and Lab Courses

Heuristic Description: *"A course and its lab cannot be taken separately."*

Explanation: The LAU regulations specify that when a course is taken for the first time, its lab must be registered with it in the same semester. Hence, advisors must ensure that the (Lab Course and lab) pair are taken concurrently. This rule of course is disregarded in the event that a student took the pair in a semester, failed one of the components, and repeats the failed component in the subsequent semester by itself.

M.H.3 Course Reachability

Heuristic Description: *"Courses are prioritized based on the highest degree of reachability."*

Explanation: A course 'degree of reachability' refers to the number of courses to which this course is a direct or indirect prerequisite. In graph theory terms, it represents the number of courses that are directly or indirectly reachable from this course. The notion of 'degree of reachability' is different from the number of neighbors, as it also includes neighbors of the neighbors as deep as the graph goes.

Consider the case of CSC243: Intro to OOP, commonly referred to as CP1. CP1 is a prerequisite to only one course CSC245: Objects & Data Abstraction (CP2), hence it has an out-degree of 1. However, CP1 is among the courses with the highest degree of reachability equal to 16. This is because taking CP2 is restricted by completing CP1, then there are many more courses that are restricted by completing CP2 including CP3, then other courses like CS electives that are in turn restricted by completing CP3 … and so on. This snowball effect is what makes CP1 a course with high priority, despite its small number of neighbors. In fact, it is interesting to note that CP1 can only be delayed by 2 semesters at most, without delaying graduation.

By ordering our courses by degree of reachability, we ensure that courses are taken as soon as possible and prioritized by order of importance.

M.H.4 Credit Limit for Semesters

Heuristic Description: *"Semesters cannot exceed their designated credit limit"*

Explanation: According to LAU rules and regulations, there is a certain limit to the maximum number of credits that can be taken within a semester. They are summarized in the table as follows.

Semester Type	Maximum Credit Limit
Fall	18 crds
Spring	18 crds
Final *	21 crds
Summer	9 crds

* Final: The last semester before graduation, it is allowed to petition for an overload of 21 credits.

M.H.5 Summer Courses Availability

Heuristic Description: *"Summer Semesters can only include courses that appear in the availability list."*

Explanation: When building a projection plan, the advisor must prevent ambiguous and risky situations like planning a course for a semester in which it might not be offered. While it is assumed that all courses are offered in all regular semesters (Spring and Fall), the list of courses offered during summers is generally more restricted, as it highly depends on the availability of instructors and the demands of students.

In order to ensure that projection plans are reliable and will need minimal changes in the long term, we use a hypothetical list of summer availability which includes courses that are typically offered in summers (LAC electives, math electives, and a few major courses). When planning for the summer semester courses, only courses appearing in this list will be included in the semester.

Computer Science Courses
CSC491: Professional Experience
Mathematics Courses
MTH207: Discrete Structures I
MTH201: Calculus III
MTH305: Probability & Statistics
Math Elective
LAC Courses
All LAC courses
Other Courses
LAS204: Technology, Ethics, and the Global Society
ENG202: Advanced Academic English
COM203: Fundamentals of Oral Communication
Free Elective

Optional Heuristics (O.H)

Optional heuristics include all of the rules that, if applied, help create a balanced and realistic plan. These heuristics, however, should not prevent the student from graduating on time, as they only exist to be practical. In this section, we will first start by describing and explaining the heuristics and how they are relaxed individually. Then, we will explain how these heuristics interact with one another, and how they are prioritized and relaxed in order to create the optimal projection plan.

O.H.1 Minimize total credits per semester

Heuristic Description: *"Avoid overloading semesters when possible."*

Explanation: This heuristic is different from the previously explained mandatory heuristic MH.4 *"Semesters cannot exceed their designated credit limit".* While the latter sets a maximum credit limit to not exceed, this optional heuristic specifies that semesters should be as light-loaded as possible. In other words, we prefer generating a semester plan that includes 6 semesters of 16 credits each, rather than 6 semesters with 18 credits in some semesters and 12 credits in others.

In our algorithm, we first try combinations of semesters with the fewest credits (typically 16 crds), if no solution is found, we increment this limit to 18 crds, one semester at a time, finally, if no solution is found despite all semesters reaching 18 crds, we allow to exceed the limit of the final semester only to 21 crds.

To sum up, the process of relaxing this heuristic goes as follows:

1. A semester with fewer credits is prioritized over a more loaded semester.
2. When generating a semester plan, we first attempt to minimize the credits for all semesters by setting their maximum load to 16 crds.
3. If no solution is found, we increase the maximum load for a single semester to 18 crds, and search again for a solution. We do this as a bottom-up process, incrementing the last semester's credit limit first.
4. We iterate over all possible combinations in a similar way as long as no solution is found. The last combination at this step would be a full load of 18 crds for all semesters.
5. If no solution was found, we increased the maximum load for the last semester to 21 crds. In LAU rules and regulations, a student must petition for this credits overload which is only allowable for the final semester. For this reason, this is an unpreferred situation that is left as a last resort and only considered if no solution is found despite a full load of 18 crds for all semesters.
6. We repeat the same steps and iterate over all combinations of 21 crds, 18 crds and 16 crds, always minimizing the number of credits, for as long as no solution is found.
7. The last attempt is relaxing the heuristic for all semesters. The resulting maximum loads to try would then be 18 crds for all semesters, and 21 crds for the last semester.

O.H.2 Minimize Additional Summers

Heuristic Description: *"Avoid including summers in the projection plan when possible."*

Explanation: This optional heuristic is quite simple to articulate. We mean that a projection with no summers is preferable to a projection with additional summers. Thus, the way we go about this heuristic is we first attempt to generate a projection plan with no summers, and if no solutions are found, we add one summer at a time while always covering all summer combinations before adding an additional summer.

O.H.3 Balancing between Different Types of Courses

Heuristic Description: *"Avoid semesters that include a heavy load of major courses when possible."*

Explanation: This heuristic aims to avoid generating a projection plan that would include unnecessarily academically challenging semesters. Its purpose is to create semesters that include a combination of different course types (LACs, major courses, minor courses ..etc) in order to alleviate the difficulty of a semester, and therefore, facilitate academic success for the student. In other terms, we prioritize creating balanced semesters that include a limited amount of major-related courses (which we assume are academically challenging). This heuristic is especially relevant to us because M.H.3 "Course Reachability" specifies that courses must be picked by order of highest reachability. Unsurprisingly, the courses with the smallest reachability degrees are often courses that are unrelated to the major. Hence, if we were to discard this heuristic, it is likely that we would end up generating highly unbalanced projection plans, with early semesters including major-related courses only, and later semesters including non-major-related courses only.

Prioritizing Heuristics

As of now, we have established what optional heuristics we will be considering for our academic advisor system. However, we still have not covered how these O.H. are prioritized in comparison

to each other. For example, do we prefer a projection with no summers but a full load in regular semesters, or do we prefer a projection plan with lighter semesters but additional summers?

This decision was subjective, and based on what we assumed students would prefer (from our experience and our peers' opinions). It goes as follows, in increasing order of importance (i.e from first to relax to last to relax).

Lowest priority (first to relax when no solution is found): O.H.3 Balancing between Different Types of courses

Average priority (next to relax when no solution is found): O.H.1 Minimize total credits per semester

Highest priority (last to relax as last attempt): O.H.2 Minimize Additional Summers

3. Generating a Projection Plan

3.1. Levelization Algorithm

A core concept in our algorithm is levelizing the graph. By levelization, we mean assigning for each node a level. The level of a node represents the earliest time a node can be taken. For example, CSC243: Intro to OOP has no prerequisites, thus it can be taken as early as possible, it is therefore of level 0. On the other hand CSC245: Objects & Data Abstraction has CSC243 as a prerequisite, thus the earliest time it can be taken is one level after the level of CSC243 (providing it has no other prerequisites), it is therefore of level 1.

For co-requisites the logic is slightly different. CSC320: Computer Organization Course has CSC245 as a co-requisite. This means that the earliest time it can be taken is at the same time as its co-requisite, thus it will be of level 1.

Assigning levels to nodes allows us to categorize them in a way that no two nodes in the same level create a conflict. This makes sense, since the formula of levelization ensures that a node with parent level x, is at least at level x+1. The exact formula actually goes as follows:

For a node n, with parent nodes 1, 2 ,3 ,....., m

$$n.level = \max \{parent_1.level+1 , parent_2.level +1, parent_3.level+1 \ldots\ldots parent_m+1 \}$$

Upon levelizing the graph, we keep track of our nodes and corresponding levels by using a Hashmap structure that maps each node to its level. We then use this structure to fill out semesters level by level, in a way that every semester includes courses from the same level only.

```java
/** Update level of a node based on the prereq/coreq relation */
public void updateLevel(Edge neighbor, Node parent) {
  if(neighbor.getNode().getVisited()) return; //if node was visited already, skip
  int current_level = neighbor.getNode().getLevel();
  // if co req
  if (neighbor.getWeight() == 0)
    neighbor.getNode().setLevel(Math.max(current_level, parent.getLevel()));
  // if pre req
  else
    neighbor.getNode().setLevel(Math.max(current_level, parent.getLevel() + 1));

}
```

Function to calculate the level of a node based on its parent. The max() ensures that the level is updated correctly through multiple iterations.

```java
/**
 * Levelize entire graph. This uses BFS over all nodes, and compute the levels.
 */
public void levelizeGraph() {
  for (Node root : nodes) {
    Set<Node> visitedNodes = new HashSet<>();
    Queue<Node> queue = new LinkedList<>();
    queue.add(root);
    visitedNodes.add(root);

    while (!queue.isEmpty()) {
      Node currentNode = queue.poll();

      List<Edge> neighbors = currentNode.getNeighbors(); // get edges with node and weight
      // System.out.println(neighbors);
      for (Edge neighbor : neighbors) {
        if (!visitedNodes.contains(neighbor.getNode())) {
          queue.add(neighbor.getNode());
          visitedNodes.add(neighbor.getNode());
          updateLevel(neighbor, currentNode);

        }
      }
    }
  }
}
```

LevelizeGraph function computes the levels of all nodes for the first time. It traverses all nodes using BFS and updates their levels with the helper function updateLevel().

```java
/**
 * This levelizes starting from a root node.
 * It only traverse nodes that are connected to the root.
 */
public void levelizefromRoot(Node root) {
  Set<Node> visitedNodes = new HashSet<>();
  Queue<Node> queue = new LinkedList<>();
  queue.add(root);
  visitedNodes.add(root);

  while (!queue.isEmpty()) {
    Node currentNode = queue.poll();

    List<Edge> neighbors = currentNode.getNeighbors(); // get edges with node and weight
    // System.out.println(neighbors);
    for (Edge neighbor : neighbors) {
      if (!visitedNodes.contains(neighbor.getNode())) {
        queue.add(neighbor.getNode());
        visitedNodes.add(neighbor.getNode());
        updateLevel(neighbor, currentNode);

      }
    }
  }
}
```

This function uses the same logic as levelizeGraph(), however, **it is only applied for updating the levels of a root node, and its dependent nodes.** It is called once a semester reached maximum load and remaining courses needs to be pushed one level down. **We use this function to avoid traversing the entire graph if it is not needed.**

```java
// Organize nodes in a hashmap accoridng to the level they belong to
public HashMap<Integer, List<Node>> computeLevelMap() {
  HashMap<Integer, List<Node>> map = new HashMap<>();
  for (Node node : getNodes()) {
    List<Node> nodesAtLevel = map.getOrDefault(node.getLevel(), new ArrayList<>());
    nodesAtLevel.add(node);
    map.put(node.getLevel(), nodesAtLevel);
  }
  return map;

}
```

Function to create the level map based on course levels

The resulting hashmap from the first levelization results in courses being categorized in 4 levels as follows:

```
Level 0: [Introduction to Object-Oriented Programming, Calculus III, Fundamentals of Oral Communications, Advanced Academic English, Technology, Ethics, and the Global Society, Basic Biology for Computer Science, Discrete Structures I, LAC Elective 1, LAC Elective 2, LAC Elective 3, LAC Elective 4, LAC Elective 5, Free Elective]
Level 1: [Objects & Data Abstraction , Computer Organization, Computer Organization Lab, Discrete Structures II, Probability & Statistics, Math Elective]
Level 2: [Algorithms & Data Structures, Database Management Systems, Operating Systems, Software Engineering, Professional Experience]
Level 3: [Parallel Programming for Multicore & Cluster Systems, Computer Networks, Capstone Project, CSC elective 1, CSC elective 2, CSC elective 3, CSC elective 4, CSC elective 5]
```

3.2. Generating Combinations of Semesters

Another key component in our code is generating semesters in the appropriate order. Manually, generating a semester is a very intuitive task: one alternates between Falls and Springs and may occasionally add a summer in between whenever needed. In code, this was a bit more challenging to implement as there are a number of constraints and calculations that need to be taken into consideration to generate a correct plan.

First, it was important to identify the semester we are starting with. This may sound trivial, but whether the projection plan starts with Fall or with Spring changes the index of the summer semester with respect to the position of the first semester.

Second, beyond correctly identifying indices, we had to cater for generating multiple combinations of semesters, where we may or may not append summers in order to try all combinations. Once all combinations are created, we pass them to the generateProjection function in this order: the first combination includes no summer, then for every new combination we increment the number of summers, until the last passed combinations include as many summers as allowed.

The way we do this is that we first calculate the number of summers that could possibly be added based on both the start semester, and the number of regular semesters left. For example, for a regular plan with 6 semesters and a FALL start, we may add at most 2 summers (one after Spring of sophomore year, and one after Spring of Junior Year). However, with a plan of 6 semesters but starting in SPRING, we can add up to 3 summers (one after Spring of sophomore year, one after Spring of Junior year and one after Spring of Senior year).

The logic of this section is implemented as static functions in Semester.java.

```java
/**
 * Function that generates the combinations of summers across the projection
 * plan. Takes k is the
 * maximum amount of summers allowed, generate combinations like [summer 1: yes,
 * summer 2: no],
 * [summer 1: no, summer 2: no] ...etc
 */
public static List<int[]> generateCombinations(int k) {
  List<int[]> combinations = new ArrayList<>();
  int[] options = { 1, 0 };

  for (int i = 0; i < Math.pow(2, k); i++) {
    int[] combo = new int[k];
    for (int j = 0; j < k; j++) {
      combo[j] = options[(i >> j) & 1];
    }
    combinations.add(combo);
    // System.out.println(Arrays.toString(combo));
  }

  return combinations;
}
```

Function to generate combinations of summers based on parameter k (maximum of summer allowed)

```java
 * @param startSeemster
 *                      is 0 for start in FALL, 1 for start in SPRING
 */

public static int getMaxSummers(int maxRegSemesters, int startSemester) {

 int max_summers;

 if (startSemester == 1 && maxRegSemesters % 2 == 0)
   max_summers = (int) (Math.ceil(maxRegSemesters / 2.0));
 else
   max_summers = (int) (Math.ceil(maxRegSemesters / 2.0) - 1);

 return max_summers;

}
```

Function to calculate the maximum amount of allowed summers. Takes as parameters the start semester, and total number of regular semesters.

```
public static List<List<Semester>> generateSemestersCombinations(int maxRegSemesters,
    int startSemester) {

  // calculate how many summers we can add based on number of semesters
  int max_summers = getMaxSummers(maxRegSemesters, startSemester);

  // generate our combinations of summers
  List<int[]> summerCombinations = generateCombinations(max_summers);
  List<List<Semester>> semesterCombinations = new ArrayList<>();

  for (int i = summerCombinations.size() - 1; i >= 0; i--)
  // for every combination of summers , starting from no summers [0,0]

  {
    List<Semester> currentCombination = new ArrayList<Semester>();
    int summersAdded = 0;
    if (startSemester == 0) // for a projection plan that start in fall
    {
      for (int j = 0; j < maxRegSemesters; j++) {

        if (j % 2 != 0) // index for a summer
        {
          if (summerCombinations.get(i)[summersAdded] == 1) {
            currentCombination.add(new Semester(name:"Summer", new boolean[] { false, false, true }));

          }
          summersAdded++;
          currentCombination.add(new Semester(name:"Fall", new boolean[] { true, false, false }));
        } else
          currentCombination.add(new Semester(name:"Spring", new boolean[] { false, true, false }));
      }
    } else if (startSemester == 1) // for a projection plan that starts in spring
    {
      for (int j = 0; j < maxRegSemesters; j++) {

        if (j % 2 != 0) // index for a summer
        {
          if (summerCombinations.get(i)[summersAdded] == 1) {
            currentCombination.add(new Semester(name:"Summer", new boolean[] { false, false, true }));

          }
          summersAdded++;
          currentCombination.add(new Semester(name:"Fall", new boolean[] { true, false, false }));
        } else
          currentCombination.add(new Semester(name:"Spring", new boolean[] { false, true, false }));
      }

    }
    currentCombination.get(currentCombination.size() - 1).setFinal(isFinal:true);
    semesterCombinations.add(currentCombination);

  }
  return semesterCombinations;
}
```

Function to generate the combination of semesters by inserting summers in the appropriate index. This function uses generateCombinations() as a helper function.

This is a sample run of the semester generation algorithm:

Max regular semesters=6, start Semester = Fall

```
Trying semester combinations for 6 regular credits, starting Fall.
Maximum amount of summers is: 2
Summer combinations are:
[1, 1]
[0, 1]
[1, 0]
[0, 0]
Semesters are :
[Fall, Spring, Fall, Spring, Fall, Spring]
[Fall, Spring, Summer, Fall, Spring, Fall, Spring]
[Fall, Spring, Fall, Spring, Summer, Fall, Spring]
[Fall, Spring, Summer, Fall, Spring, Summer, Fall, Spring]
```

A sample run for generating all combinations of semesters for inputs k=6 regular semesters, i=0 start in Fall.

Max regular semesters=6, start Semester = Spring

```
Trying semester combinations for 6 regular credits, starting Spring.
Maximum amount of summers is: 3
Summer combinations are:
[1, 1, 1]
[0, 1, 1]
[1, 0, 1]
[0, 0, 1]
[1, 1, 0]
[0, 1, 0]
[1, 0, 0]
[0, 0, 0]
Semesters are :
[Spring, Fall, Spring, Fall, Spring, Fall]
[Spring, Summer, Fall, Spring, Fall, Spring, Fall]
[Spring, Fall, Spring, Summer, Fall, Spring, Fall]
[Spring, Summer, Fall, Spring, Summer, Fall, Spring, Fall]
[Spring, Fall, Spring, Fall, Spring, Summer, Fall]
[Spring, Summer, Fall, Spring, Fall, Spring, Summer, Fall]
[Spring, Fall, Spring, Summer, Fall, Spring, Summer, Fall]
[Spring, Summer, Fall, Spring, Summer, Fall, Spring, Summer, Fall]
```

A sample run for generating all combinations of semesters for inputs k=6 regular semesters, i=1 start in Spring.

3.3. Iterative search for a solution and Heuristic Relaxation

-After having established our level map for all our graphs, we need to start assigning courses to semesters for the student.

1. We start off by calculating reachability for all nodes: it calculates how many other nodes each node in a set can reach within a graph. It uses DFS starting from each node to explore all reachable nodes, tracking these in a HashSet. After completing the DFS for a node, it records the number of nodes that can be reached.

```
 */
public void computeReachability() {
  for (Node node : nodes) {
    Set<Node> visited = new HashSet<>();
    DFS(node, visited);
    node.setReachability(visited.size() - 1);
  }
}
```

2. We loop through all the combinations of semesters, and then for every semester we get the credit limit appropriate, this is determined by the function “determineCreditLimits” based on the different factors represented by the params passed to it

```
// if requirements aren't finished but there's still semesters to add
if (currentSemester < semesters.size())

{
        // for this currentsemester and level

        int currentMaxCrds;
        int currentCredits;
        int currentMajorRelatedCredits;
        int[] creditsHeuristic = determineCreditLimits(semesters.get(currentSemester),
                        allowExtraCreditsInLastSemester);
```

```
    /**
     * Function to determine the different credits allowed in a semester based on
     * different factors
     * like type of semester (summer or not), last semester (final or not) ...etc.
     * return an int[]
     * with the different allowed values.
     */
    private static int[] determineCreditLimits(Semester semester,
                    boolean allowExtraCreditsInLastSemester) {
            if (semester.getUserCredits() > 0) {
                    return new int[] { semester.getUserCredits() };
            }
            if (semester.isSummer()) {
                    return new int[] { SUMMER_SEMESTER_CREDIT_LIMIT };
            } else if (semester.isFinal() && allowExtraCreditsInLastSemester) { // this for the last semester when
                                                                                // we are
                    return new int[] { FINAL_SEMESTER_CREDIT_LIMIT }; // allowing 21 credits.
            } else if (!semester.isFinal() && allowExtraCreditsInLastSemester) { // this is when we are directly
                                                                                 // passing 18
                    return new int[] { REGULAR_SEMESTER_CREDIT_LIMIT }; // for all semesters.
            } else {
                    return new int[] { RECOMMENDED_CREDIT_LIMIT, REGULAR_SEMESTER_CREDIT_LIMIT };
            }
    }
```

-The first condition checks if the user has specified the number of credits for that specific semester, in this case, it returns the credits set for that semester.

-The second one checks if the semester is a summer, and returns the summer credit limit

-Third check if the semester is a final semester and if we are allowing 21 credits.

-Fourth checks for the case when we are allowing 21 credits but when the semester is not final. This is particularly important because since we already exhausted all possibilities and there was no solution, instead of passing 16 and trying all the combinations from scratch we are directly passing to it 18.

-The last check is in the regular case of a normal semester we are passing an array that we will be looping over that consists of the regular credit limit of 16 and the maximum one of 18.

3. For the current semester, We start by looping through our heuristics, we have an outer loop that will represent the major credits heuristic, and that first starts by allowing the balancing. After that, we proceed with the course credit limit.

```
// non major courses
boolean[] majorBalancingHeuristic = { true, false };
for (boolean allowBalancing : majorBalancingHeuristic) {
        for (int i : creditsHeuristic) {
```

4. We then treat courses level by level, we first sort the level based on reachability and then loop through the courses one by one:

```
// sort by reachability
HashMap<Integer, List<Node>> map = graph.computeLevelMap();
List<Node> coursesToConsider = map.get(currentLevel);
ComparatorTool comparator = new ComparatorTool();
comparator.setStrategy(s:"major");
Collections.sort(coursesToConsider, comparator);
comparator.setStrategy(s:"reachability");
Collections.sort(coursesToConsider, comparator);
```

5. We loop through the courses that we have in the current level and while we are adding courses to the semester, we must handle the interesting case of prerequisites.

It is worth noting that while levelization ensures that there is no conflict between prerequisites within the same level (since a course and its prerequisite can never be assigned to the same level), this is not always the case when it comes to prerequisites. In fact, as explained in the levelization algorithm, a course and its prerequisite are of the same level. For example, CSC320: Computer Organization Course and CSC245: Objects & Data Abstraction are both initially assigned to level 1. This is no issue as long as both are taken within the same semester, and no issue in the case where CSC230 is deferred to a later semester. However, an edge case arises if we were to, for some reason, skip CSC245 and pick CSC230. This situation is unlikely to happen, mostly because a course would generally have a lower priority than its prerequisite (reachability of CSC230 < reachability of CSC245), but it could happen in certain scenarios.

For example, consider the case of the summer semester of sophomore year and assume we are selecting courses from level 2, which include CSC375: Database Management Systems and CSC491: Professional Experience for which CSC375 is a prerequisite to CSC491. We have already established that both of these courses exist in the same level since they hold a prerequisite relationship, and we are iterating over courses within this level to select them for our summer semester in accordance with the heuristic of the summer course availability. It happens that CSC375 is not included in the summer availability list, thus it is skipped and we move to the next course. We then reach CSC491 which on the other hand appears on the list, thus it is selected. Here, we have stumbled upon an edge case in which a course was selected despite its prerequisite not being selected.

Similar situations can arise due to the summer availability heuristic and during customization, when users specify the courses they want to take for the upcoming semester. Thus, we need to handle the issue of prerequisites within our loop.

For this, we implement a checker hasCompletedCoreq() that verifies if the current course has already completed its prerequisites (either the prerequisite has been taken in previous semesters, or has already been picked in the current semester). This allows us to proceed safely with the course selection and avoid this edge case swiftly.

```java
/**
 * This function searches for the corequisites of a course and determines if the
 * corequisites have been
 * taken or not. if the corequisites have bene visited but not taken, then we
 * cannot take this course.
 * Else, we can take it, so we move on with the rest of the code.
 *
 * @param node:        is the node we are deciding on
 * @param noesAtlevel: is the list that contains all courses in that same level.
 *                     Rememebr that corequsiites are suppsoed to be on the same
 *                     level, so we do not want to look further than that
 */
private static boolean hasCompletedCoreq(Node node, List<Node> nodesAtLevel, List<Node> coursesAtSemester) {
        List<Course> corequisites = node.getCourse().getCoreq();
        // if node has no corequisite, do nothing
        if (corequisites.isEmpty()) {
                return true;
        }
        // course has coreqs
        for (Node n : nodesAtLevel) {
                // if we foudn the coreq in the same level, and it was visited but not taken,
                // then skip
                if (corequisites.contains(n.getCourse())) {
                        if (n.getVisited() && !coursesAtSemester.contains(n))
                                return false;
                }
        }
        return true;
}
```

```java
// starts filling semester level by level
List<Node> coursesAtCurrentSemester = new ArrayList<>();
for (int j = 0; j < coursesToConsider.size(); j++) {
        Node n = coursesToConsider.get(j);
        // first check if my node needs processing
        if (n.getVisited()) {
                continue;
        }
        n.setVisited(b:true); // set it to processed as if we are about to start
                              // processing

        // handle summer list unavailability
        if (semesters.get(currentSemester).isSummer()
                        && unavailableSummerCourses.contains(n.getCourse()))
                continue;

        if (!hasCompletedCoreq(n, coursesToConsider,
                        coursesAtCurrentSemester))
                continue;
```

6. While we are looping throughout the semester we make sure that the number of credits taken does not exceed the maximum credit limit set. In addition we check if the major credit heuristic is on and if yes, we make sure it is taken into consideration:

```
if (currentCredits + n.getCourse().getCrds() <= currentMaxCrds) {
        // allow balancing is true, means we only add a course if, it's
        // either not a major
        // course, or it's major but major limit wasn't reached yet
        // allow balancing is false, means we add anyway so if statement
        // must pass everytime
        boolean isMajor = n.getCourse().isMajor();

        if (!allowBalancing || (!isMajor
                        || allowBalancing && currentMajorRelatedCredits
                                        + n.getCourse().getCrds() <= MAJOR_CREDITS_SEMESTER_LIMIT)) {
```

7. We make sure that every lab is taken with its course, we keep track of the number of major and total credits taken.

```
{
        Node labNode = findLabNode(n,
                        coursesToConsider);
        if (currentCredits + n.getCourse()
                        .getCrds()
                        + labNode.getCourse()
                                        .getCrds() <= currentMaxCrds) {
                coursesAtCurrentSemester.add(n);
                coursesAtCurrentSemester
                                .add(labNode);
                n.getCourse().setisCompleted(
                                isCompleted:true);
                labNode.getCourse()
                                .setisCompleted(
                                                isCompleted:true);

                semesters.get(currentSemester)
                                .setNodesAtSemester(
                                                coursesAtCurrentSemester);
                currentCredits += n.getCourse()
                                .getCrds()
                                + labNode.getCourse()
                                                .getCrds();
                currentMajorRelatedCredits += n
                                .getCourse()
                                .getCrds()
                                + labNode.getCourse()
                                                .getCrds();

        }
        // Prevent lab/course from being
        // processed again
        labNode.setVisited(b:true);
```

8. If the course is not a lab, we increase the total credits taken and major credits if it is the case:

```
} else // no lab issue
{
        coursesAtCurrentSemester.add(n);
        n.getCourse().setisCompleted(isCompleted:true);

        semesters.get(currentSemester)
                        .setNodesAtSemester(
                                        coursesAtCurrentSemeste
        currentCredits += n.getCourse()
                        .getCrds();
        if (isMajor)
                currentMajorRelatedCredits += n
                                .getCourse()
                                .getCrds();
```

9. After we are done with level and we assigned all the courses possible to the appropriate semesters. We push down all the left courses to the lower level and then we levelize the whole graph one more time.

```
// Handling courses that were not picked for this current semester.
for (Node course : map.get(currentLevel)) {
        if (!semesters.get(currentLevel).getNodesAtSemester()
                        .contains(course)) {
                course.setLevel(course.getLevel() + 1); // push one level down
                course.setVisited(b:false);

        } else {
                course.setVisited(b:true);
        }
}

for (Node course : map.get(currentLevel)) {
        if (!semesters.get(currentSemester).getNodesAtSemester()
                        .contains(course)) {
                graph.levelizefromRoot(course); // push all levels below
                map = graph.computeLevelMap();
        }
}
```

-This process is repeated through each level until we reach a solution and complete all the courses and assign them to a semester. If there is no solution, we start relaxing the heuristic just like explained previously.

3.4. Customizing the Projection Plan

In most cases, students usually customize their course projection plan throughout their academic journey. However, this needs to be achieved while still abiding by the rules and, if possible, while still graduating on time. Our system offers flexibility to the student to test different possibilities and customize their projection plan as they like with three features: (1) The ability to specify summer preferences, (2) The ability to specify a maximum credit limit per semester, (3) The ability to pick a list of courses to include for the upcoming semester.

- **Specifying summer preferences:** The user is able to customize his projection plan by specifying what summers to enroll in, if any.

  This is done by allowing the user to choose among all combinations of summer preferences, and generate the projection plan based on the chosen input. The implementation behind this feature relies for the most part on the different functions for generating semester combinations explained in Section 3.2: Generating combinations of semesters. Once the user picks their preferred summer combination, the full semester combination (which includes Spring and Fall semesters) is generated, and is directly passed as input to the main function findSchedule() which will now only consider this unique semester combination instead of iteratively trying different ones.

- **Specifying the maximum number of credits per semester:** The user is able to specify a maximum credit limit to apply to a specific semester. This feature is especially useful when a user wants a specific semester to have lower load. For example, a user might want their last semester of senior year to have a maximum load of 12 crds in order to complete an internship for employment after graduation, or to have time to apply for graduate school.

- **Specifying courses to take in the upcoming semester:** The user is able to specify the courses they wish to take for the upcoming semester. This feature is aimed to simulate the real advising procedure during registration period, where students submit an advising form in which they specify courses they wish to take for the upcoming semester, and advisors assess the choices, and either approve or disapprove the advising form.

4. The Academic Advisor System

4.1. Initialization and Serialization of Objects

In order to save objects, specifically our list of courses, and user files, we use Java Object serialization to convert the object from instance in memory to byte streams that can be stored. We then reconstruct the object through deserialization as needed.

This was done by enabling classes to implement the Serializable Interface, then calling the functions ObjectOutputStream and ObjectInputStream for reading and writing.

Finally, helper functions were used to handle the file reading and writing. Files were given a specific extension, for example User objects were serialized into files with a ".us" extension. The initialization of users and courses can be found in the Init Classes.

```
//saves user as a file . file name is the user's ID.us
public static File saveUser(User user)
{
  String file_name= user.getID()+".us";
  File file = new File(file_name);
  try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(file)))
  {
    oos.writeObject(user);
    return file;
  }catch(IOException e)
  {
    System.out.println("Error creating User File");
    return null;
  }

}
```

Function to serialize the user object into a file with a .us extension

```java
public static User loadUser(File file)
{

  if (file.exists())
  {
    try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(file)))
    {
      return (User) ois.readObject();
    }
    catch (ClassNotFoundException e)
    {
      e.printStackTrace();
      return null;
    }
    catch (IOException e)
    {
      e.printStackTrace();
      return null;
    }
  }
  else
  {
    System.out.println("No such user exist");
  }
  return null;
}
```

Function to deserialize a user file and return the user

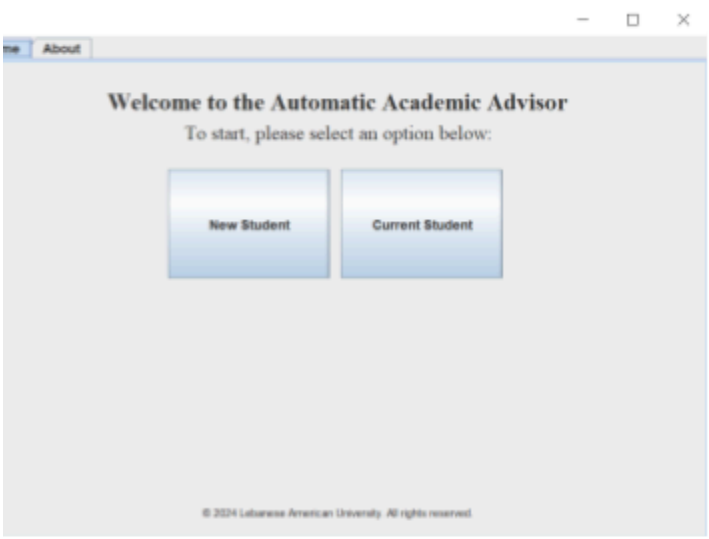
4.2. Graphical User Interface

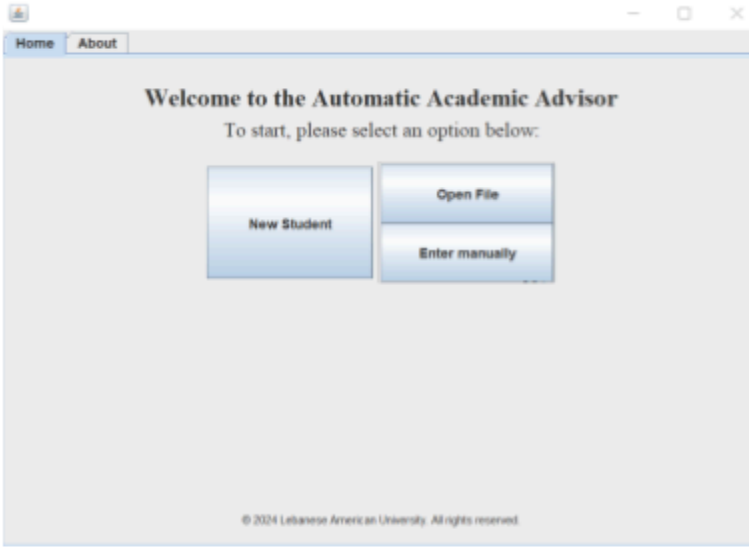
In order to display the results of our algorithm, we have opted for a simple GUI interface using Java Swing Components. The interface of our project is meant to be simple and straight to the point and it includes a landing window that prompts the user to enter his data.

We distinguish three cases:

- **A student who is new to the platform, and newly enrolled [New Student]:** Is meant for students who have not undertaken any semesters yet, and do not have their data saved yet.

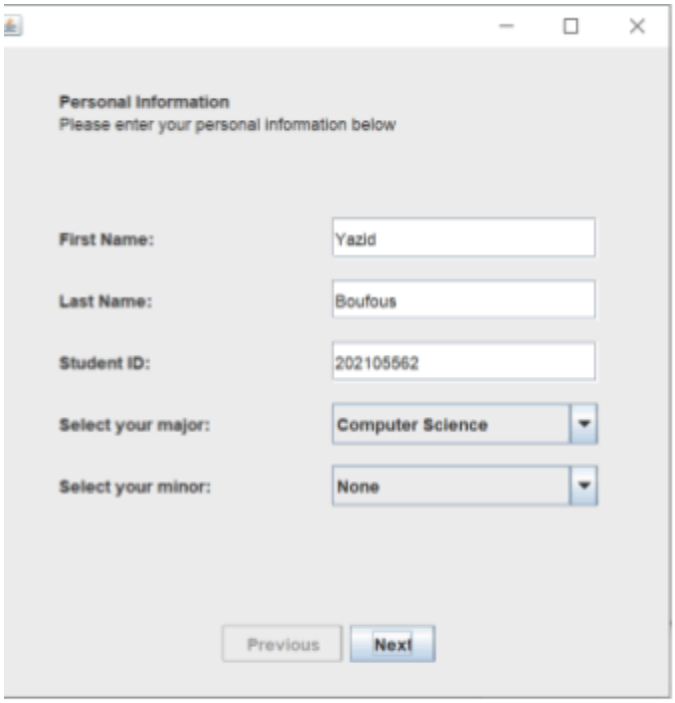
- **A student who is new to the platform, but previously enrolled [Current Student → Enter Manually]:** Is meant for students who are already enrolled and have already completed semesters, but do not have their data saved.
- **A student who is not new to the platform [Current Student →Open File]:** Is meant for any student who already has their data saved in their unique user file.

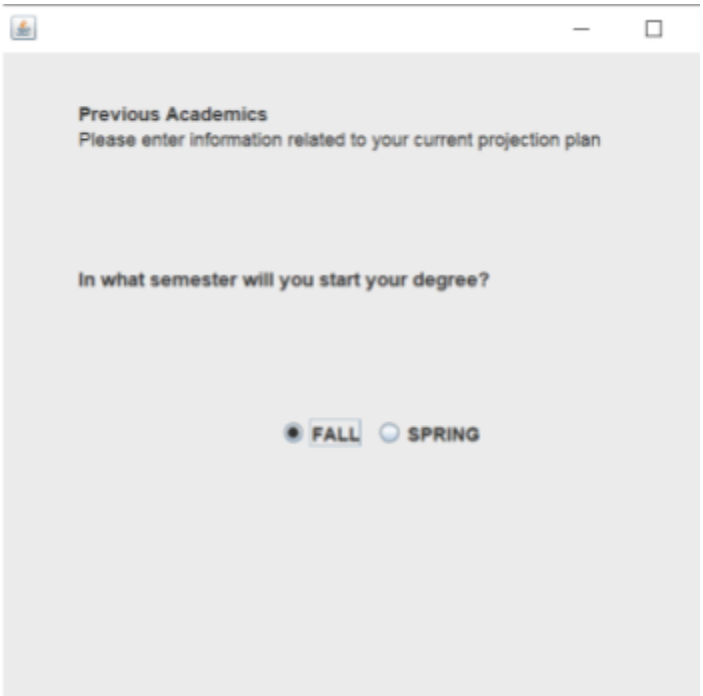




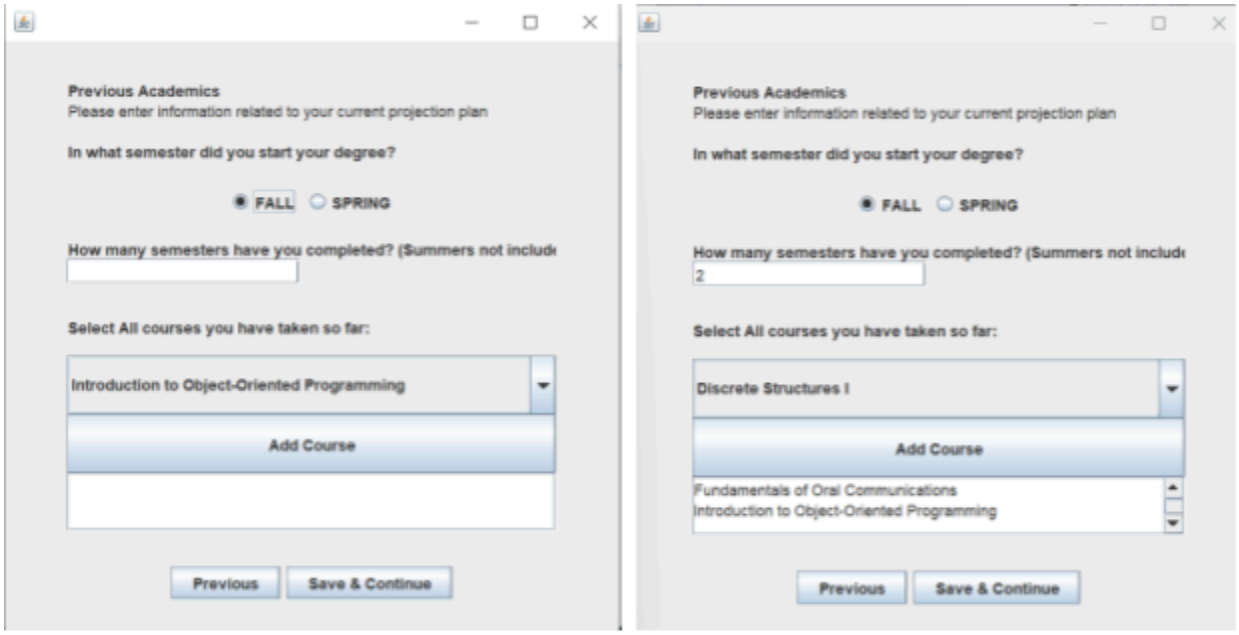
Landing window of the Automated Academic Advisor with options to start.

When we create a user for a new student or an already enrolled student manually, they will be asked to enter their basic personal information, as well as specifying their start semester.





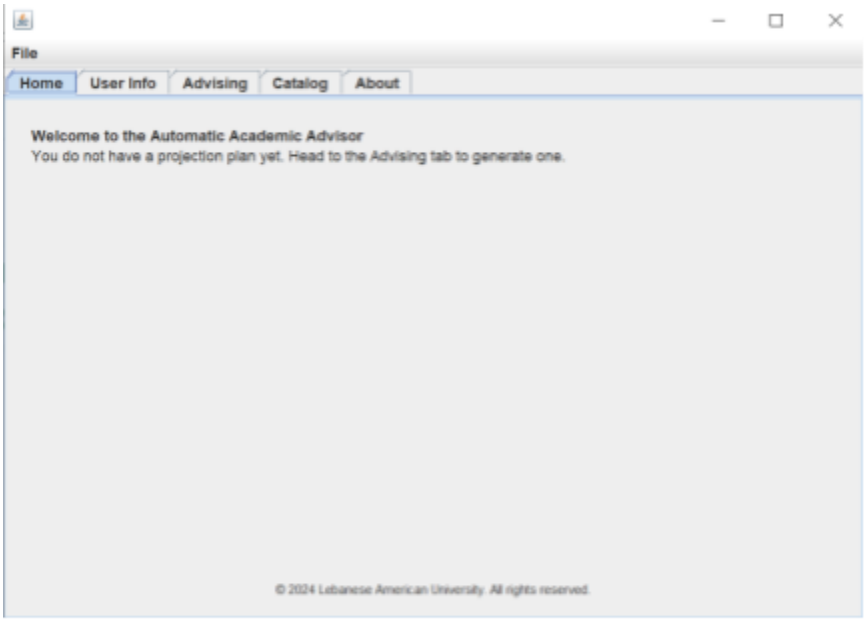
If the user is already enrolled in the institution (current student), he will be asked to enter additional data related to the semesters he completed, such as the number of semesters, and the list of completed courses.



Previously enrolled students (Current Student) are prompted to enter additional information about their completed academics.

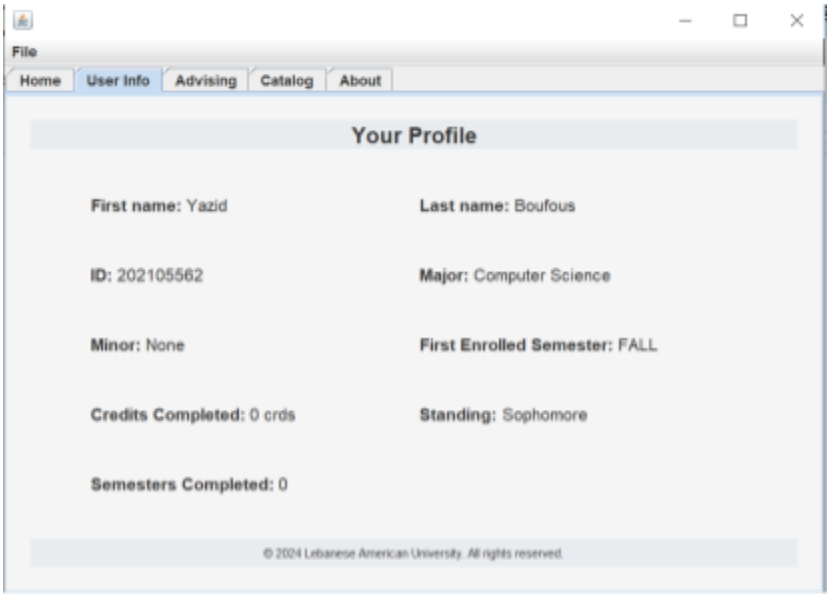
The application consists of 5 tabs: Home tab, User info, Advising, Catalog and About.

- **Home tab:** When the user is new and does not have a saved course projection plan, it will display a message that asks the user to generate a graduation plan from the advising tab. This tab will be different when we are loading a user and opening their file. It will display the latest saved generated projection plan.



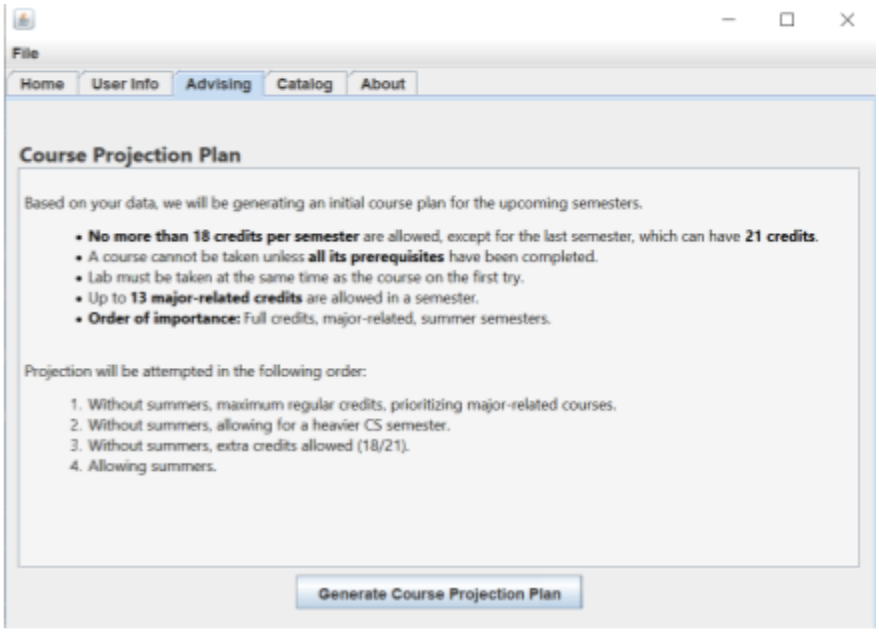
Home Tab for a new student.

- **User Info tab**: This tab contains all the necessary information about the student, such as name, ID, major and minor, first enrolled semester, number of completed credits, semesters completed, and the standing which is decided based on the number of completed credits according to LAU.



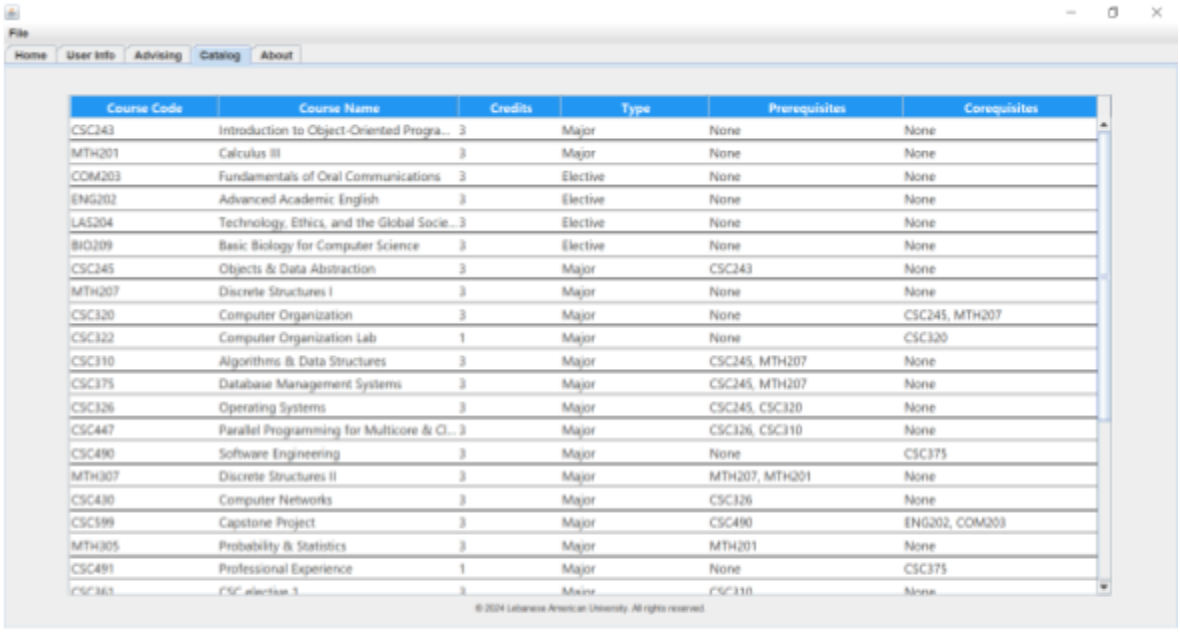
User Info Tab for a newly enrolled student

- **Advising tab**: This tab is dedicated to everything that has to do with the advising, it will be used to generate the projection plan based on the user data. It displays some details about how the projection plan is done and the rules followed for the user to refer to.



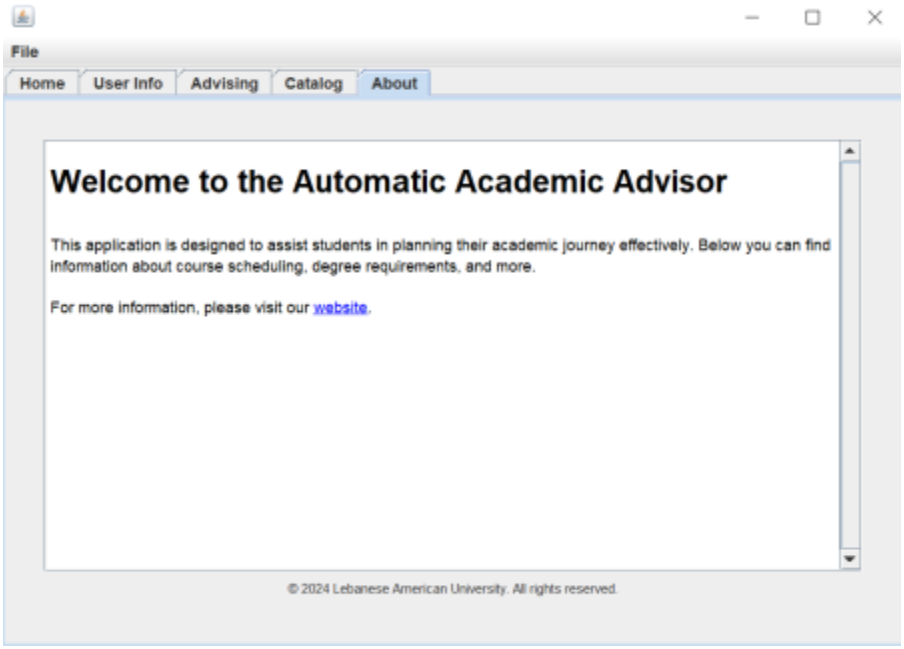
Advising tab for a new student, it contains instructions on projection plan generation.

- **Catalog tab:** It contains the catalog of the computer science degree at LAU, with the number of credits and type of every course, and the prerequisites and corequisites.



Catalog Tab, which includes information about all courses in the CS Degree.

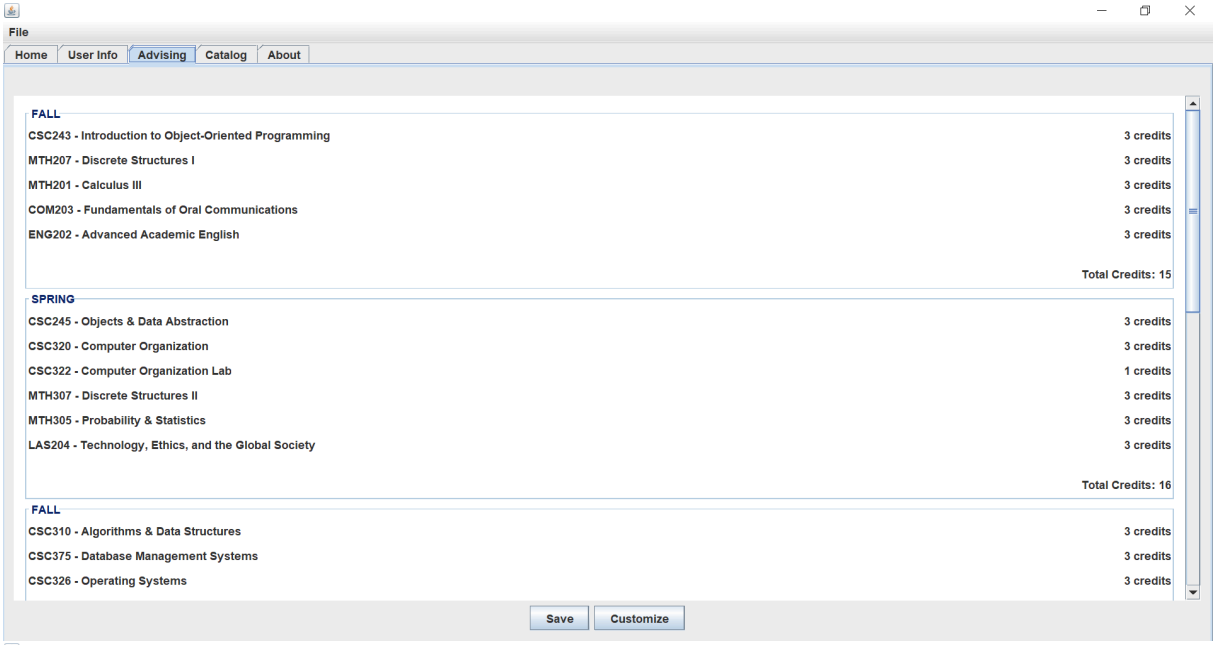
- **About tab:** a brief description of the app.

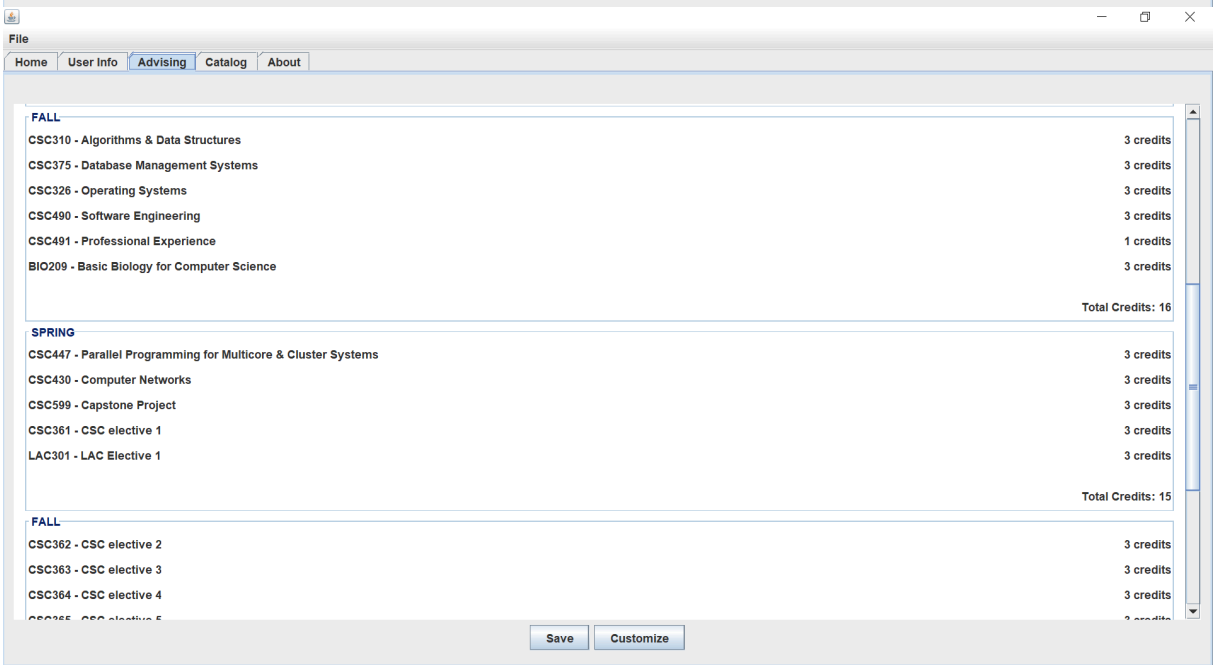


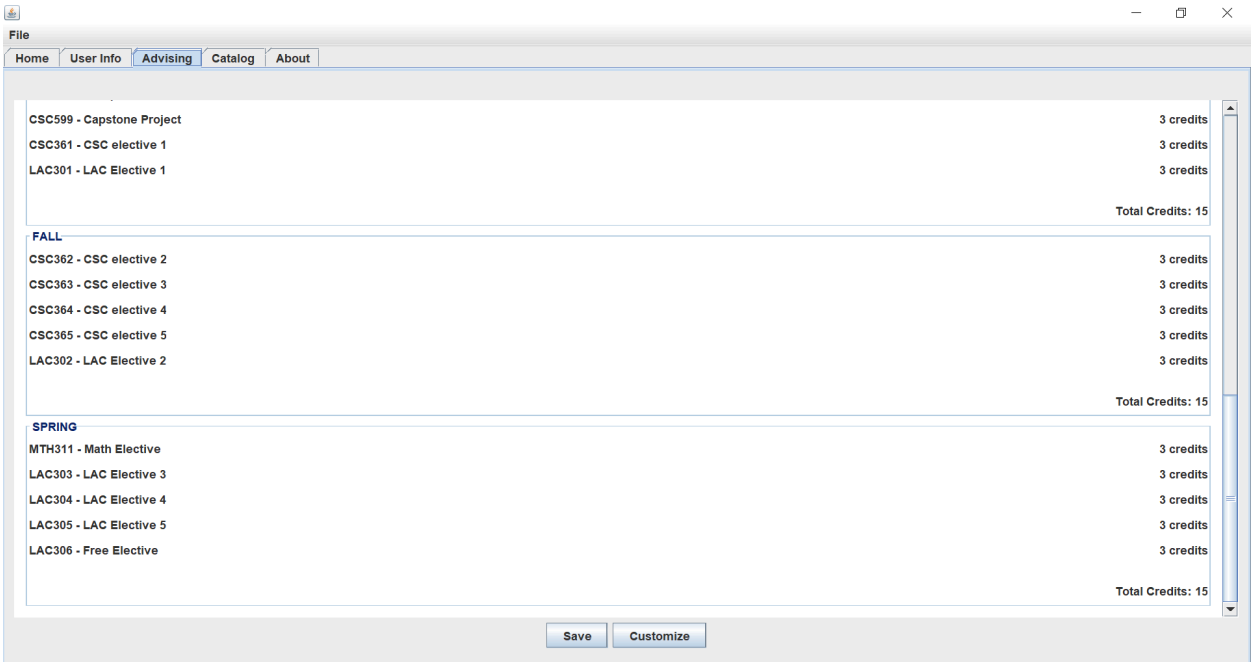
5. DEMO

- In this section, we will see different scenarios for generating the course plan:

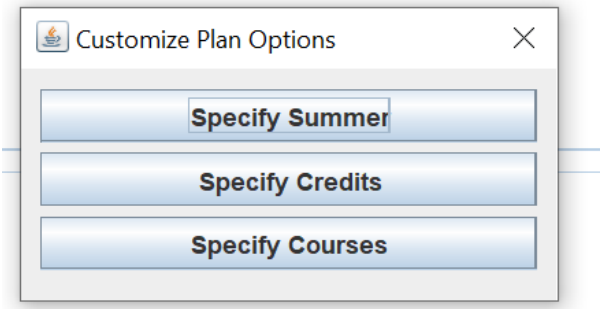
- New student: 0 completed courses (Spring semester): In every semester, the recommended credit number per semester 16 was also respected. Since the courses can be taken in 6 regular semesters, there were no summer semesters added. The major course credit 13 was also taken and the lab is taken along with the course in the same semester.

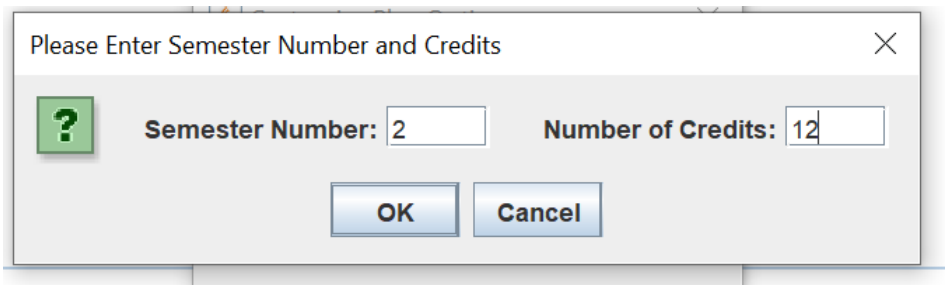


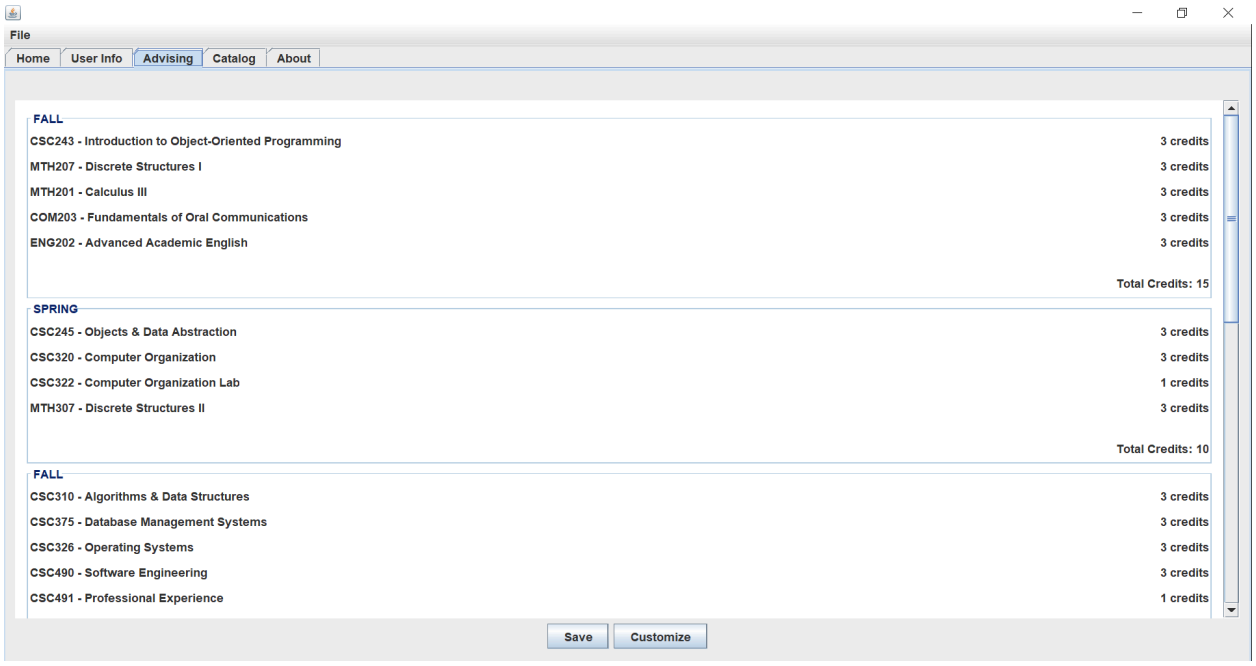


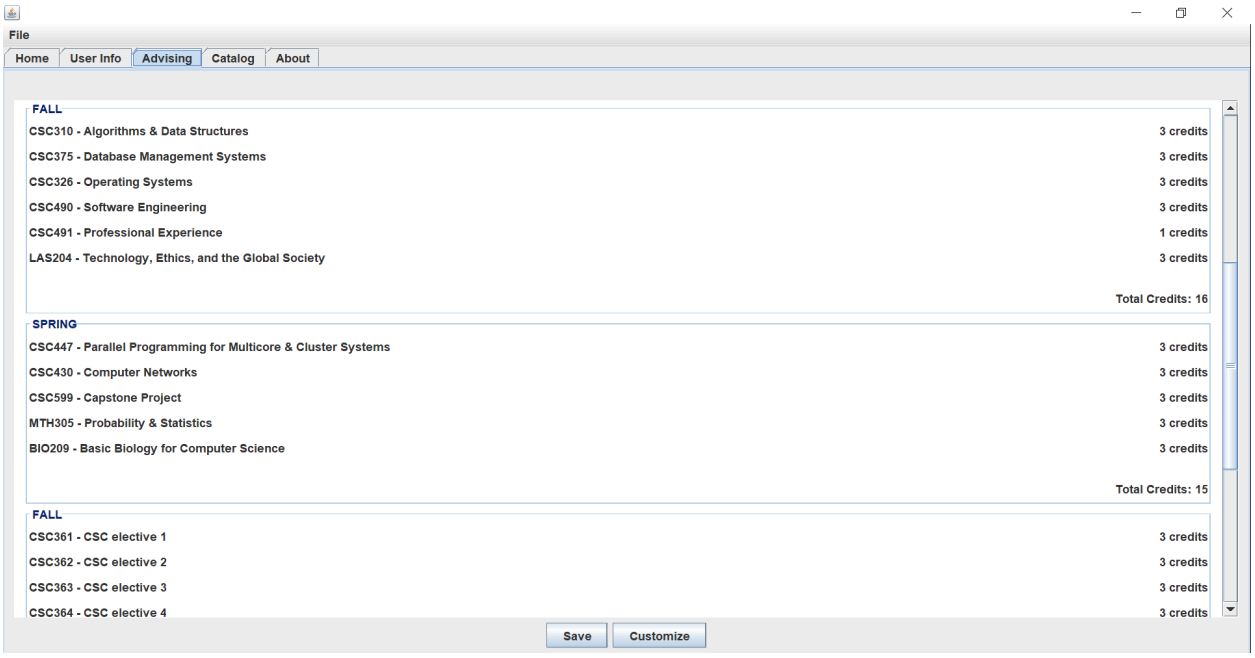


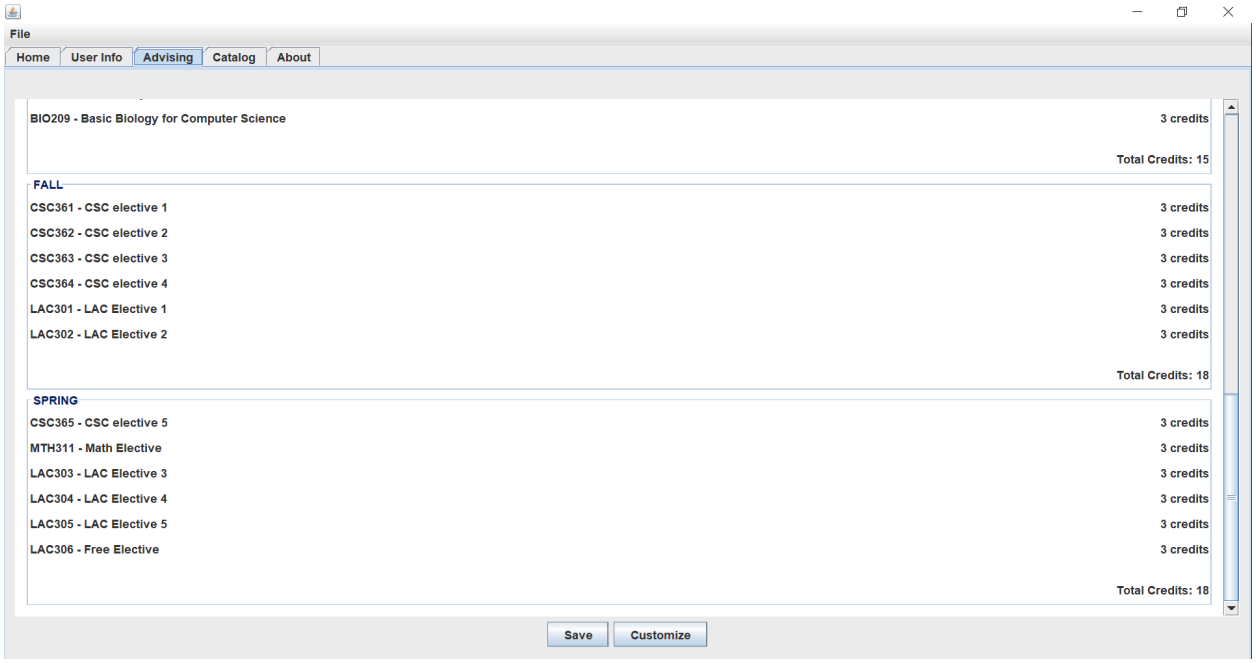
- Let us specify credits in a specific semester: the max credit allowed for the spring semester is 12, and we will push all the courses left to the next levels. This lead to increasing the number of credits in the last 2 semesters to 18 credits.



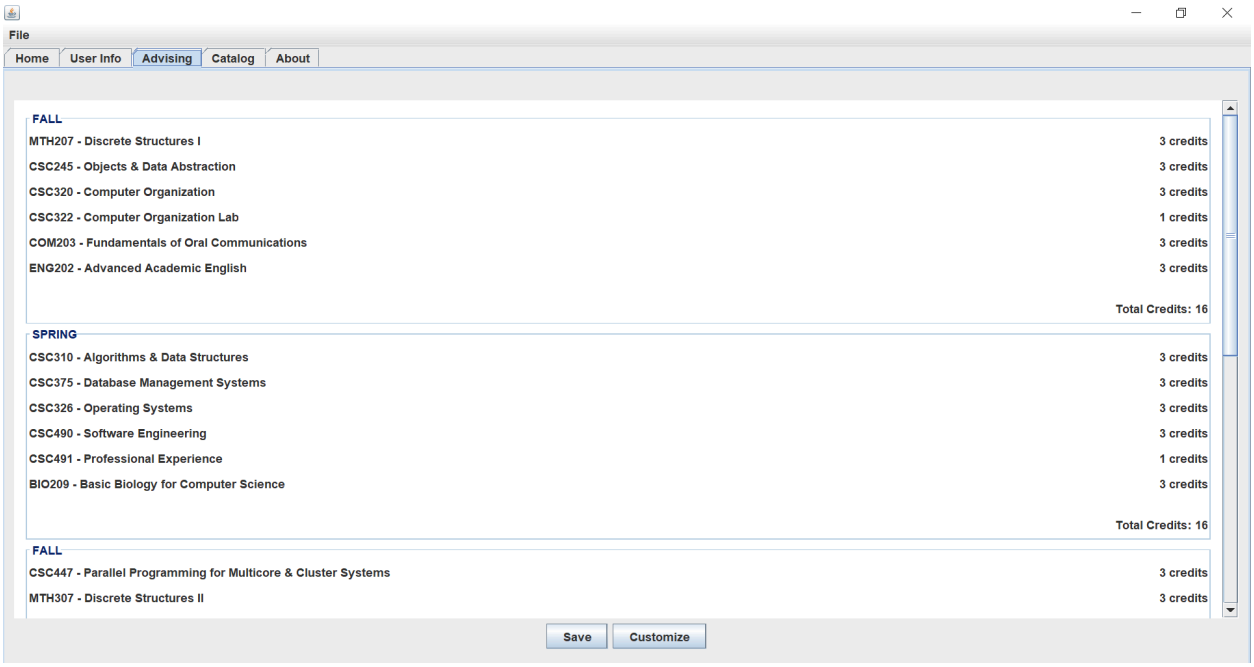


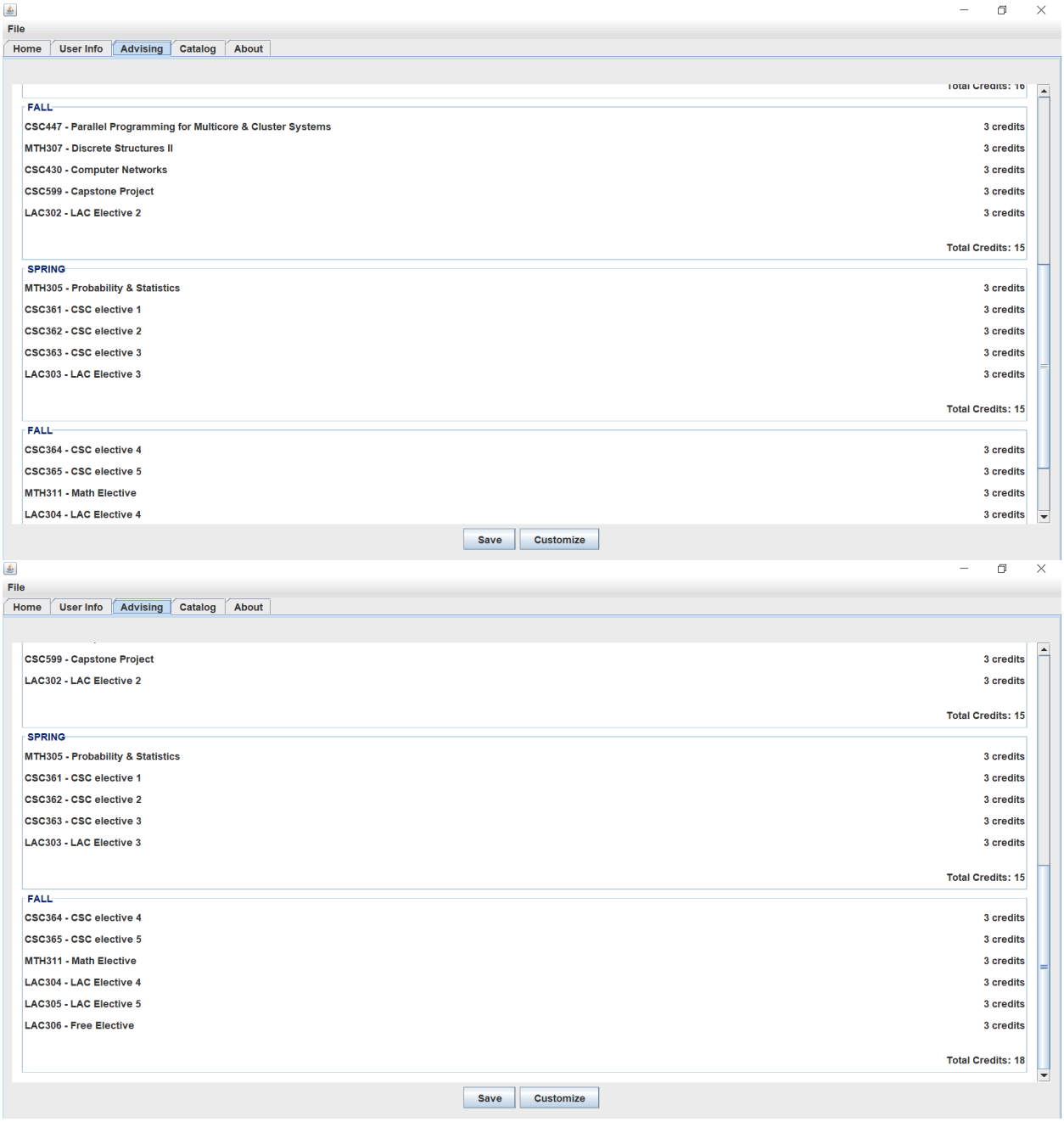

File
Home
User Info
Advising
Catalog
About
FALL
CSC243 - Introduction to Object-Oriented Programming 3 credits
MTH207 - Discrete Structures I 3 credits
MTH201 - Calculus III 3 credits
COM203 - Fundamentals of Oral Communications 3 credits
ENG202 - Advanced Academic English 3 credits
Total Credits: 15
SPRING
CSC245 - Objects & Data Abstraction 3 credits
CSC320 - Computer Organization 3 credits
CSC322 - Computer Organization Lab 1 credits
MTH307 - Discrete Structures II 3 credits
Total Credits: 10
FALL
CSC310 - Algorithms & Data Structures 3 credits
CSC375 - Database Management Systems 3 credits
CSC326 - Operating Systems 3 credits
CSC490 - Software Engineering 3 credits
CSC491 - Professional Experience 1 credits
Save
Customize


File
Home
User Info
Advising
Catalog
About
FALL
CSC310 - Algorithms & Data Structures 3 credits
CSC375 - Database Management Systems 3 credits
CSC326 - Operating Systems 3 credits
CSC490 - Software Engineering 3 credits
CSC491 - Professional Experience 1 credits
LAS204 - Technology, Ethics, and the Global Society 3 credits
Total Credits: 16
SPRING
CSC447 - Parallel Programming for Multicore & Cluster Systems 3 credits
CSC430 - Computer Networks 3 credits
CSC599 - Capstone Project 3 credits
MTH305 - Probability & Statistics 3 credits
BIO209 - Basic Biology for Computer Science 3 credits
Total Credits: 15
FALL
CSC361 - CSC elective 1 3 credits
CSC362 - CSC elective 2 3 credits
CSC363 - CSC elective 3 3 credits
CSC364 - CSC elective 4 3 credits
Save
Customize

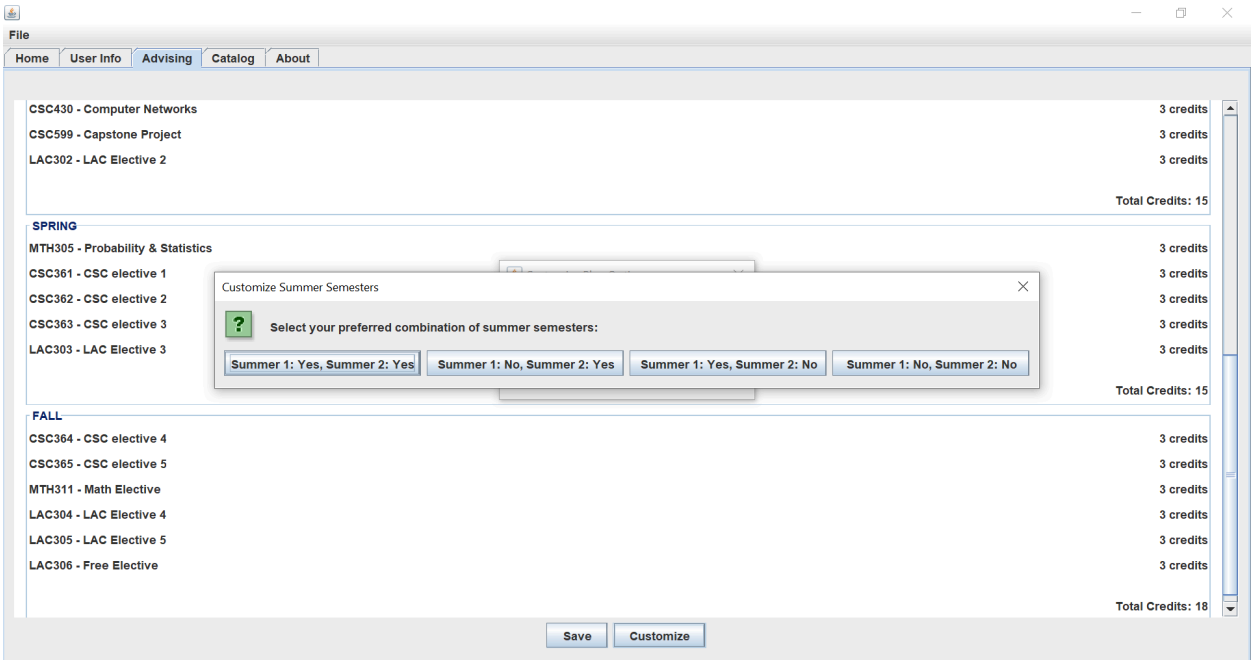


- Let us take another case of a student who has completed 1 semester with 12 credits and who started in Spring: The system displays the projection plan with all 5 upcoming semesters.

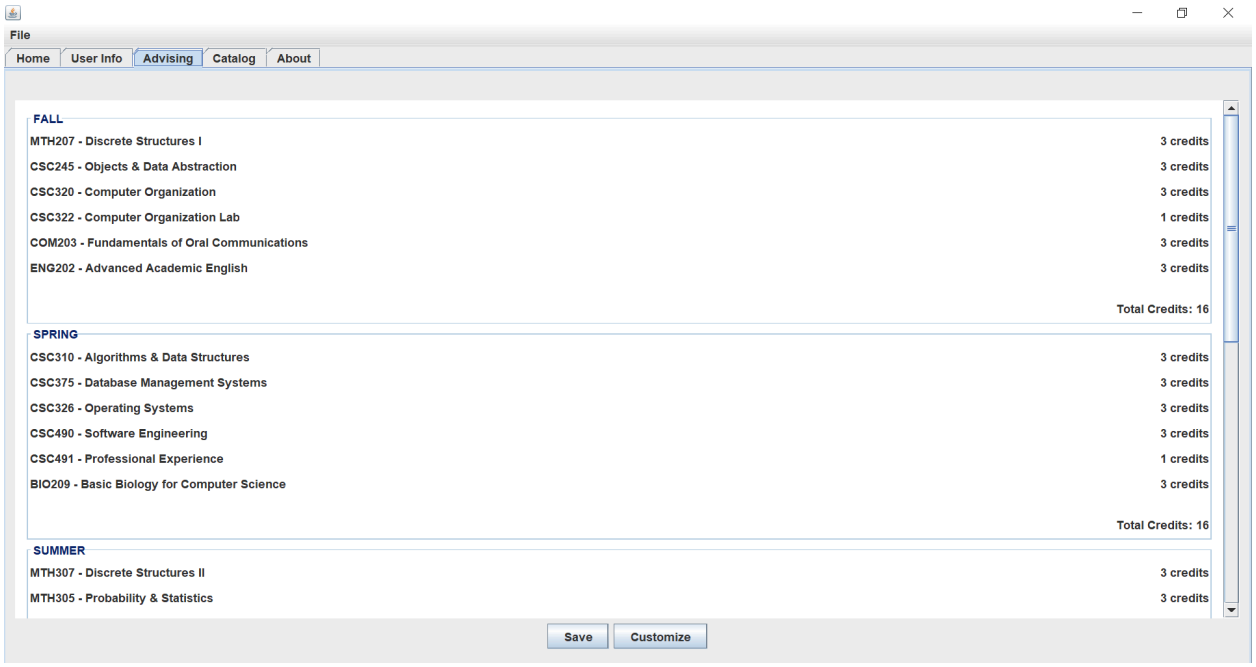


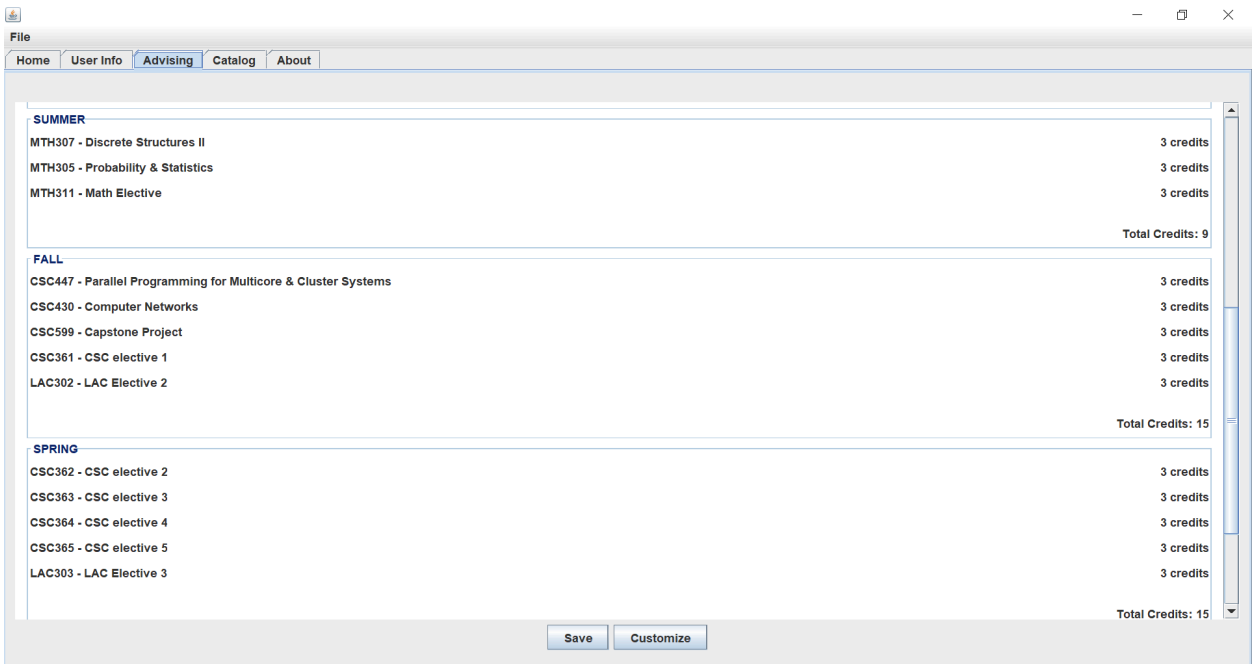


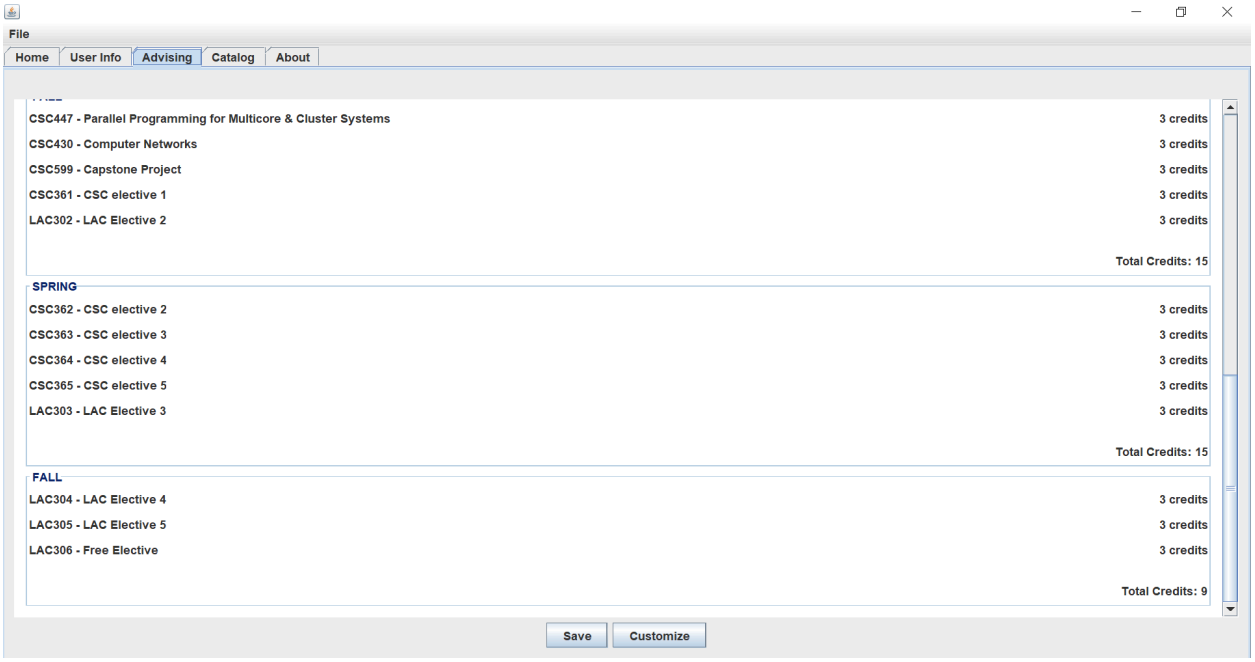
- We will allow the student to specify summers: This will display to the user all the possible summer combinations that the user can choose from.



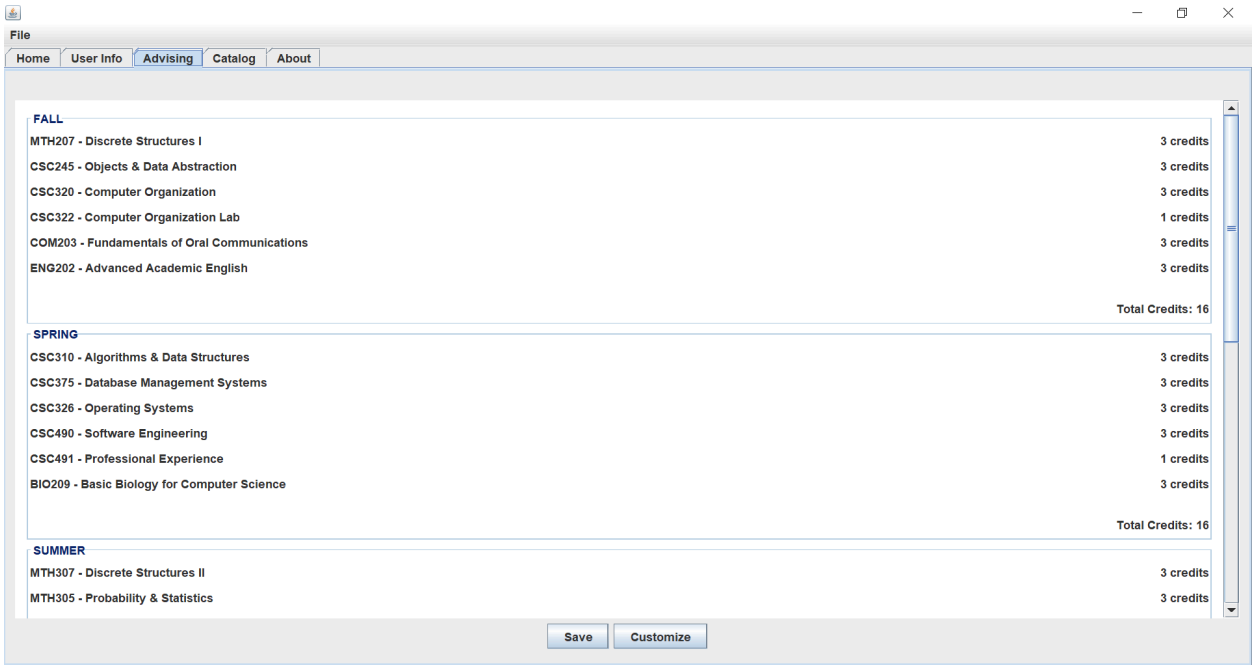
-The case for Summer 1: Yes, Summer 2: No: an additional summer semester was added in the first year, and all the unavailable courses in summer were excluded. The courses are being picked based on the heuristics set.

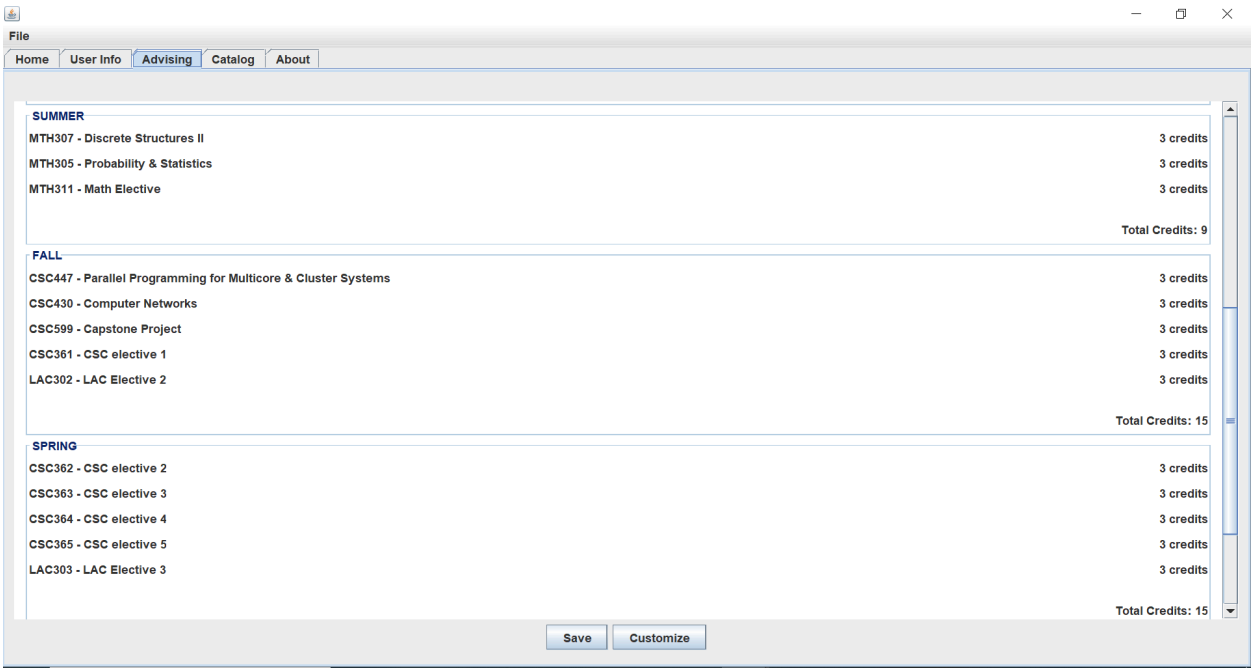


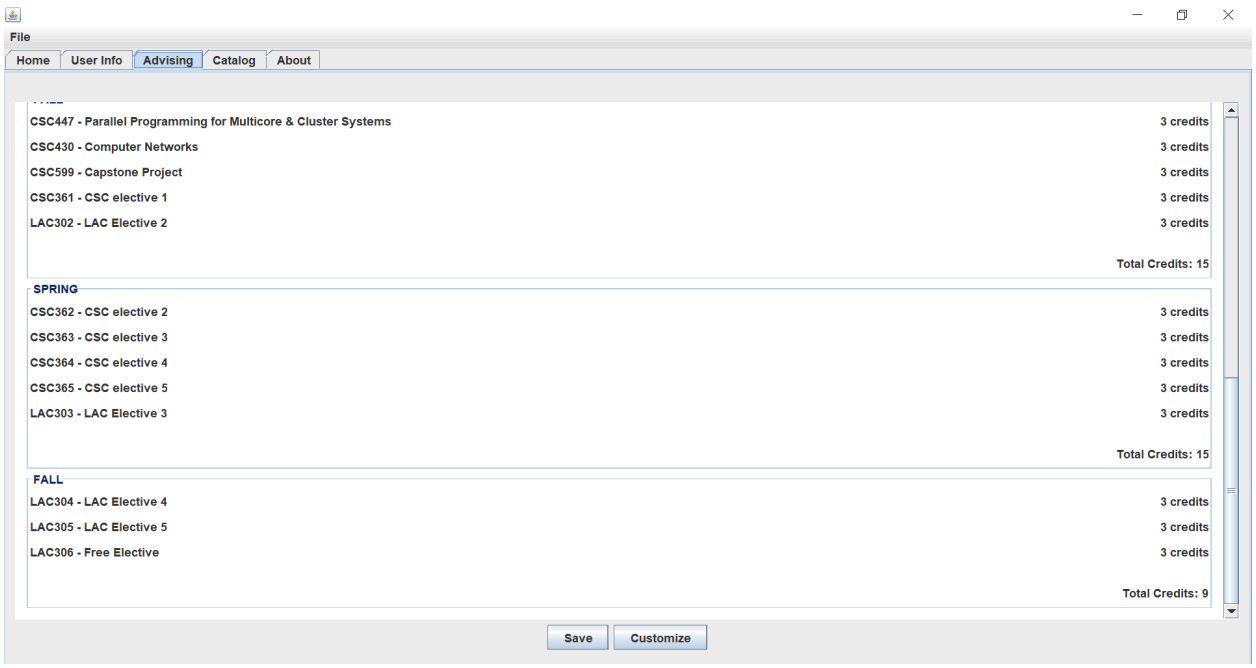




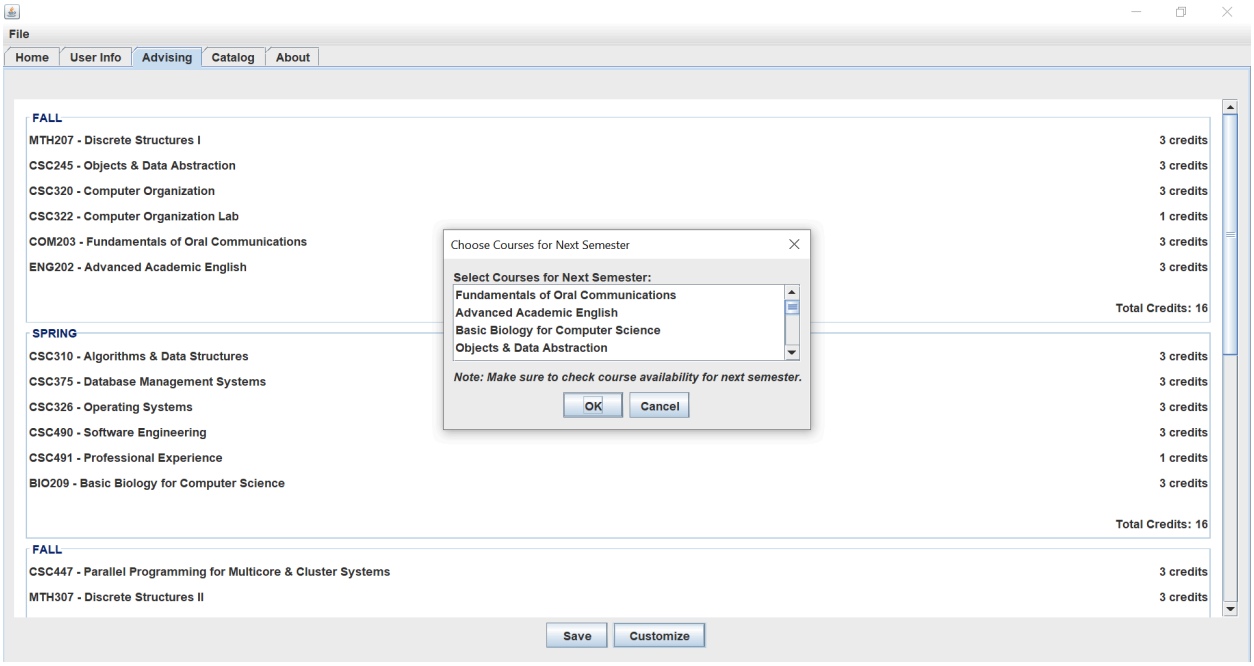
-The case for summer 1 No and summer 2 Yes: an additional summer semester was added in the second year, similarly all the unavailable courses in summer were excluded. The courses are being picked based on the heuristics set.

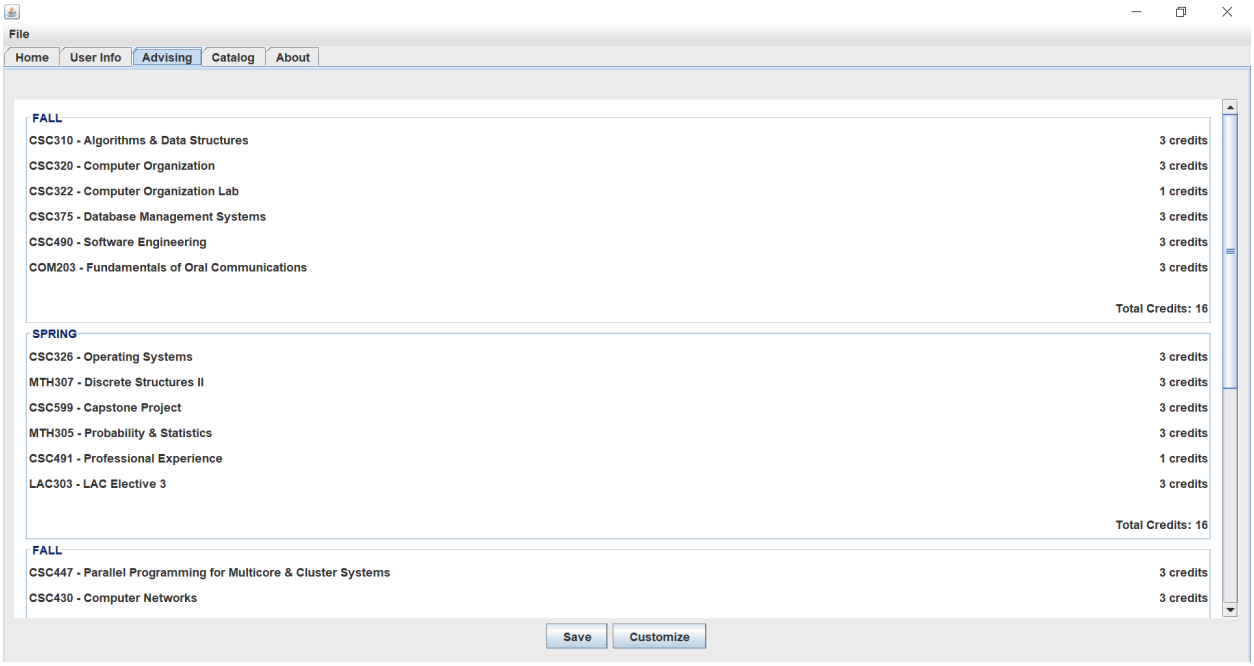

File
Home
User Info
Advising
Catalog
About
FALL
MTH207 - Discrete Structures I 3 credits
CSC245 - Objects & Data Abstraction 3 credits
CSC320 - Computer Organization 3 credits
CSC322 - Computer Organization Lab 1 credits
COM203 - Fundamentals of Oral Communications 3 credits
ENG202 - Advanced Academic English 3 credits
Total Credits: 16
SPRING
CSC310 - Algorithms & Data Structures 3 credits
CSC375 - Database Management Systems 3 credits
CSC326 - Operating Systems 3 credits
CSC490 - Software Engineering 3 credits
CSC491 - Professional Experience 1 credits
BIO209 - Basic Biology for Computer Science 3 credits
Total Credits: 16
SUMMER
MTH307 - Discrete Structures II 3 credits
MTH305 - Probability & Statistics 3 credits
Save
Customize


File
Home
User Info
Advising
Catalog
About
SUMMER
MTH307 - Discrete Structures II 3 credits
MTH305 - Probability & Statistics 3 credits
MTH311 - Math Elective 3 credits
Total Credits: 9
FALL
CSC447 - Parallel Programming for Multicore & Cluster Systems 3 credits
CSC430 - Computer Networks 3 credits
CSC599 - Capstone Project 3 credits
CSC361 - CSC elective 1 3 credits
LAC302 - LAC Elective 2 3 credits
Total Credits: 15
SPRING
CSC362 - CSC elective 2 3 credits
CSC363 - CSC elective 3 3 credits
CSC364 - CSC elective 4 3 credits
CSC365 - CSC elective 5 3 credits
LAC303 - LAC Elective 3 3 credits
Total Credits: 15
Save
Customize



-The user can also specify the courses that they can take for the next semester. We display to the user only the courses that they are allowed to take for the next semester to avoid any conflicts. It displays the semesters starting for the next semester.





6. Limitations

It is important to note that our Automated Academic Advisor has a number of limitations. First, this advisor system only caters for the Computer Science Degree Requirements at LAU. While we believe that the core concepts behind our advisor system are robust, and that the modular implementation of our code allows it to easily be expanded to other majors, it is worth noting that new heuristics might need to be added to handle eventual unseen cases of other degree requirements.

Second, as we are working with multiple heuristics and conditions for both course picking and graph levelization, it is important to carefully assess the run time complexity of our algorithm and optimize it to avoid redundant or unnecessary operations. For example, we recommend optimizing the graph levelization to only update the level of the nodes if needed, and quickly identify if delaying a course will lead to no solution by keeping track of the maximum allowed delay. This optimization would allow us to directly reject a situation that requires delaying a

course further than its allowed maximum delay limit, as we know for a fact that it will not lead to a solution within the graduation deadline without having to go through all iterations.

Finally, we acknowledge that this automated advisor is still unable to handle many tasks the way a real advisor does and many more features to enhance this system can be added to improve its flexibility. Some features we find interesting to add would be the ability to add or delete a minor, to change majors, and to select courses to include in subsequent semesters.

The above limitations highlight areas of future improvement to address in future work. We believe that this project, with further work and enhancements, has the potential to become a fully fledged academic advisor system that could be implemented in academic institutions to bring innovation and practicality to the current existing advising process.

7. Conclusion:

In conclusion, the Automated Academic Advisor system will play a significant role in the field of academic planning and advising. By leveraging graph theory and heuristics, this system efficiently handles complex course scheduling challenges, enabling personalized and optimized academic pathways for students. Although currently tailored to the Computer Science degree requirements, the principles and architecture of this system provide a robust foundation for expansion to other majors and minors. Overall, this project not only fulfills the immediate academic needs of students but also sets the stage for future innovations in academic advising, making it a valuable tool for educational institutions aiming to enhance student retention and success.

8. Codebase:

- The code is available in the below GitHub repository link:

  Automated-Academic-Advisor/Automated-Virtual-Academic-Advisor at main · yazidboufous/Automated-Academic-Advisor (github.com)