Course Selection System Project Report

Zi Hu, Houmin Sun, Yuju Weng

Abstract

This project developed a system to assist DKU students in automatically generating course selection plans. The system uses a database to store course data, including time slot, prerequisites, anti-requisites, and the graduation requirements that each course satisfy and stores information inputted by students, including majors, completed courses, and expected future courses. The system generates course schedules for the next and future semesters based on input information, ensuring that there are no conflicts in terms of time, prerequisites, and anti-requisites. The course plan for the future semester involves topological sorting algorithms. This system can provide personalized course selection plans, improving course selection efficiency.

1 Introduction

1.1 Background

The Course Selection Database is designed to assist students at DKU in managing and planning their academic schedules efficiently. This software will support students in selecting courses that align with their academic goals while addressing various constraints, such as prerequisites, anti-requisites, and scheduling conflicts. By leveraging this database, students will receive a customized course plan that accounts for both the institution's academic requirements and individual preferences.

1.2 Function

Application Domain: This software falls within the academic advising and course scheduling domain, integrating data and rules from DKUHUB and providing a user-centered interface. Its primary role is to facilitate complex scheduling, ensuring that students meet their program requirements and optimize their semester schedules without conflicts.

Objectives and Goals: The system will assist students in arranging courses in the following areas:

1. Avoid time conflicts.
2. Managing prerequisite and anti-requisite requirements.

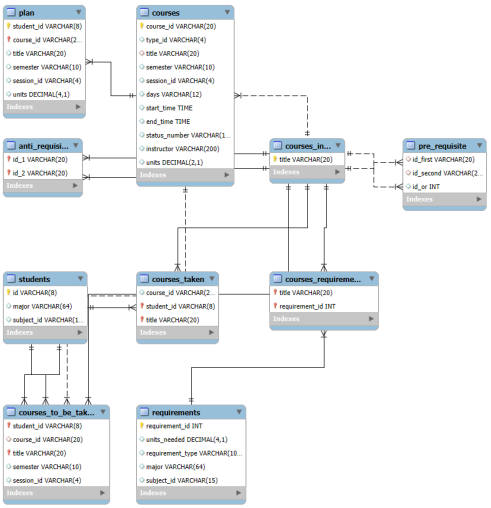

Fig. 1 Relation Schema

3. Planning a clear path to meet all credit and graduation requirements.

4. Meet other personalized needs of the users such as time periods and instructors of the courses.

Functional Boundaries: The Course Selection Database will generate course plans based on data from DKUHUB, including student input on course interests, previously completed courses, and personal availability. This tool will not act as a definitive advisor but rather as a scheduling assistant. While it will provide recommendations, students may need to consult academic advisors for final confirmation on critical academic decisions.

2 Database Schema

2.1 Courses

- course_id: The primary key for courses, it is generated and unique, the syntax is 'Semester Year number'
- type_id: Indicating course types(lab, recitation, lecture)
- title: The name of the course
- other attributes: They are used to filter classes and determine the order of having courses.

2.2 Courses_info

This table has only one attribute 'title', and it is used to make constraints to other 'title' attributes.

2.3 Pre_requisite and Anti_requisite

- id_1 and id_2: Indicating that course id_1 is an anti-requisite or pre-requisite to course id_2
- id_or: Some pre-requisite requirements will require taking course 1 or course 2. This attribute is a marker to show this relationship

2.4 Requirements

Since there are different types of requirements in DKUHUB for students to graduate, and each of them has different syntax, they are unified as the units needed for a requirement and the courses it includes.

- requirement_id: The primary key for requirements.
- unit_needed: The units needed to meet the requirements
- other attributes: To make classification and categorization of requirements

2.5 Course_requirements

This table has two attributes, 'title' and 'requirement_id', and it is for building relationships between courses and requirements.

2.6 Students

- id: This attribute is the primary key
- major and subject_ id: These two attributes are used to filter the courses that a student may take

2.7 Courses_taken

This table is used to indicate the relationship between students and the courses they take.

2.8 Courses_to_be_taken

This table is used to connect students with the courses they choose to take

- semester and session_id: Indicating the period of certain courses

2.9 Plan

This table is to contain the results of the project. Its attributes include keys that show its belonging and the course information it has.

2.10 ER model

Our database is designed for making course selection plan. The two most important entity sets in the ER model represents are "**courses**" and "**students**", including different attributes like "**section**" and "**major**" to make filter and matching. Other

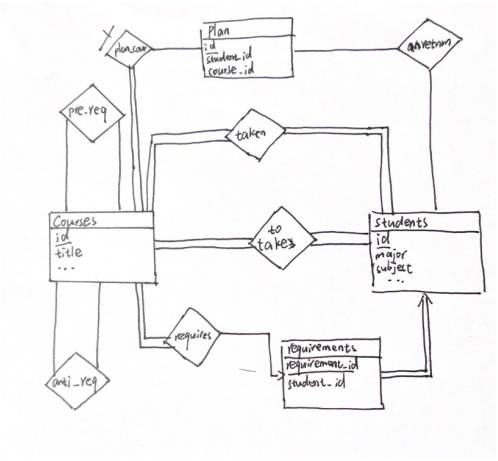

Fig. 2 ER-model

relationship sets and entity sets are necessity to make filter and constraints in all aspects.

The "requirements" set represents all the requirements to be satisfied for a student to graduate and get a degree of their target major. All the requirements are unified into several attributes, the course it includes and the unit to satisfy it. These requirements are categorized and connecting to "**courses**" with another relationship set "**courses_requires**".

The anti-requisite courses and pre-requisite courses are designed into two relationship sets that make constraints within courses. The attributes in these sets represents the contrary and the order of courses. Besides, "**taken**" and "**to be taken**" sets are designed to target the courses chosen by the user and constraint according to the courses that have been taken.

3 Important Functions and Implementations

3.1 Course Selection For One Semester

3.1.1 Basic Methods

Two steps must be implemented to make a course selection plan for one semester. One is the enumeration of courses that crafts all the possible permutations and combinations of them, and the other is creating a method to check whether it is valid.

The 'Courses' table will first be filtered according to the user's intention input from the interaction interface, and a recursive function will enumerate through it to store possible course lists. The 'check' function will consider all the constraints, removing those with time conflicts and that don't satisfy the pre-requisite requirements.

This function is mainly implemented in Python, as it constantly builds temporary tables and enumerates with a large scale of data.

3.1.2 Implementation of Time Conflicts

A few different methods were implemented due to the variation in the data scale that will be processed.

When the total number of courses in the course list is small, the best algorithm is to check the combination of every two courses in the course list by this expression: $courseA.starttime <= courseB.endtime and courseB.starttime <= courseA.endtime$.

The expected time complexity for this method is $O(n^2)$

```
for course in filter_list:
                    for course2 in filter_list[filter_list.
                        index(course)+1:]:
                        if course['title'] == course2['
                            title'] and course['units'] > 0
                             and course2['units'] > 0:
                            return
                        if not any(day in ['Mo','Tu','We','
                            Th'] for day in [d for d in [
                            course['days'], course2['days'
                            ]] if all(x not in d for x in [
                            'Mo','Tu','We','Th'])]):
                            continue
                        if course['start_time'] <= course2[
                            'end_time'] and course2['
                            start_time'] <= course['
                            end_time']:
                            return
```

Another algorithm is to implement the method above. When the data scale is too large and the number of courses in the course lists exceeds 10^2, transferring the timeline into the index and checking the time periods on the timeline will be a better choice.

```

def time_Check(course_list):
    time_line = [[0 for _ in range(100)] for _ in range
        (4)]
    for course in course_list:
        start_time = course['start_time'] / 900
        end_time = course['end_time'] / 900
        days = course['days']
        if 'Mo' in days:
            time_line[0][start_time] = 1
            time_line[0][end_time + 1] = -1
        if 'Tu' in days:
            time_line[1][start_time] = 1
```

```
                time_line[1][end_time + 1] = -1
            if 'We' in days:
                time_line[2][start_time] = 1
                time_line[2][end_time + 1] = -1
            if 'Th' in days:
                time_line[3][start_time] = 1
                time_line[3][end_time + 1] = -1
        for i in range(4) :
            for j in range(100):
                time_line[i][j] += time_line[i][j - 1]
                if (time_line[i][j] > 1):
                    return False
        return True
```

The data type that records the time in MySQL is 'TIME', and when it is fetched in Python, it will become an integer that equals the number of seconds. It will be divided by 900 to become the time point in a day as we consider every 15 minutes as a time point. The number of courses that took place at a certain time point will be recorded in the list 'timeline'. By using 'difference array' and 'suffix sum' algorithms, the process of recording courses on the timeline will be accelerated: $timeline[starttime] += 1, timeline[endtime + 1] += 1$. Calculating the prefix sum of the timeline will get the number of courses at each time point. If there is more than 1 course at the same point, this course list is not valid. The expected time complexity of this algorithm is $O(lengthoftimeline)$. In our project, the length will be 96.

In the comparison of these two algorithms, the better choice is the first one as there will only be 3 courses per session.

3.1.3 Implementation for Enumeration

The enumeration is implemented through a recursive function that has the parameters of 'course list' and 'units'. Each layer of the function will add one course to the course list and update the total units of the course list. When the total unit has reached the expected amounts, it will call the 'check' function and insert checked course lists into a list recording all the qualified combinations.

The brute-force algorithm for this is to enumerate every course in the course list for every layer. Since the total number of courses in one session will not exceed 3, and the target courses for a single student will be limited to a few majors, this algorithm can meet the requirement. Its expected time complexity is $O(n^3)$, and n's order of magnitude will not exceed 100.

```
    def list_courses(filter_list ,unit ,session):
            if unit == 10 or unit == 8 or unit == 6:
                check(filter_list ,session)
                return
            elif unit > 10:
                return
            for course in courses_list:
```

```
                    Adding the lab course and rec courses
                        together
                    list_courses(filter_list + [course,
                        course_lab, course_rec], unit+course[
                        'units'], session)
                return
```

The expected number of combinations will not exceed 10, because the courses that belong to one major are quite limited, and the constraints between courses are strict enough. This has led to an enormous waste of efficiency as the qualified course list is much less than the scale of enumeration. It will also burden the 'check' algorithm as the 'check' function will have to check all the requirements.

```
    for row in pre_requisite[course1]:
                            if row['id_first'] in
                                courses_taken1:
                                judge_or = True
                            if pre_requisite.index(row)
                                == len(pre_requisite)
                                -1 or row['id_or'] !=
                                pre_requisite[
                                pre_requisite.index(row
                                )+1]['id_or']:
                                if judge_or == False:
                                    judge = False
                                    break
                                judge_or = False
                        for row in anti_requisite[
                            course1]:
                            if row['id_1'] in
                                courses_taken2:
                                judge = False
```

This is checking all the anti-requisite and pre-requisite requirements for a single course.

To accelerate the enumeration, the thought of topological sort can be applied to ensure only courses that are not constrained by the requisition requirements are added to the course list during recursion. The queue in topological sort will be replaced by a set that records all the valid courses, and each recursion will go through the whole set to find all the combinations. Because the set is implemented through the hash table in Python, this algorithm will not add any time complexity with the operations of the set.

```
def function listcourse(filter_list, units, session)
    if units == 10 or units == 8 or units == 6:
                check(filter_list, session)
                return
```

```
            if units > 10:
                return
            for course in sett:
                if course['units'] > 0 and course['
                    session_id'] == session:
                    judge = True
                    sett.remove(course)
                    recovery = set()
                    for courses in graph_in[course['
                        title']]:
                        graph_in_degree[courses] -= 1
                        if graph_in_degree[courses] ==
                            0:
                            sett.add(course_dict[
                                courses])
                            recovery.add(course_dict[
                                courses])
                    for course2 in conflict[course['
                        course_id']]:
                        if course2 in sett:
                            sett.remove(course2)
                            recovery.add(course_dict[
                                courses])
                    adding correspond rec and lab
                        courses as well
                    listcourses(filter_list+[course,
                        course_lab+course_rec],units+
                        course['units'],session)
                    for course_R in recovery:
                        sett.add(course_R)

            return
```

This algorithm is much faster as it makes reduction to the number of enumerations. If the time conflicts are also considered during pruning, the whole algorithm will be accelerated again as it only considers valid course lists.

```
    class TimeTable:
    def __init__(self):
        #                    1000                    ,
                    0
        self.time_line = ['' for _ in range(100)]

    def add_course(self, start_time, end_time,days):
        #                  ,
                        1
```

```
8
9           start_time = start_time / 900
10          end_time = end_time / 900
11          for i in range(start_time, end_time):
12              if 'Mo' not in self.time_line[i] and 'Mo'
                    in days:
13                  self.time_line[i].append('Mo')
14              if 'Tu' not in self.time_line[i] and 'Tu'
                    in days:
15                  self.time_line[i].append('Tu')
16              if 'We' not in self.time_line[i] and 'We'
                    in days:
17                  self.time_line[i].append('We')
18              if 'Th' not in self.time_line[i] and 'Th'
                    in days:
19                  self.time_line[i].append('Th')
20
21      def has_conflict(self, start_time, end_time,days):
22          start_time = start_time / 900
23          end_time = end_time / 900
24          for i in range(start_time, end_time):
25              if (('Mo' in self.time_line[i] and 'Mo' in
                    days) or
26                  ('Tu' in self.time_line[i] and 'Tu' in
                        days) or
27                  ('We' in self.time_line[i] and 'We' in
                        days) or
28                  ('Th' in self.time_line[i] and 'Th' in
                        days)):
29                  return True
30          return False
31
32      def clear(self):
33          #
34          self.time_line = [0 for _ in range(1000)]
```

This is an implementation of the time conflicts solving algorithm mentioned in the long-term solutions sector in Python.

3.1.4 Overview of this function

The total time complexity for course selection in one semester will be $O(enumeration) \times O(check)$. The worst theoretical complexity is $O(n^3)$ since the time complexity for checking is only a constant due to the dataset.

However, combining all the optimization mentioned and making choices according to the dataset can accelerate it to approximately $O(answerset)$, because all the

qualified combinations will only be enumerated once, and the 'check' function can be removed in that case.

3.2 Course Selection For Long Term

3.2.1 Basic Methods

The Long term schedule concludes all the classes need to be taken in a certain major. Help them have a basic understanding of which class is better to take early to avoid potential pre-requisite problems. Instead of only taking 1 level classes and then 2 level classes. In order to implement this function, the difficult part is to find the course taking order to avoid any pre-requisite and anti-requisite problems. For these two relationships, we found that the number of pre-requisites is relatively large, and most non-100-level courses have pre-requisites, while the number of anti-requisites is quite small. Therefore, the complexity bottleneck is mainly on pre-requisite, which is the part we need to focus on optimizing.

3.2.2 Use Topological Sort to avoid pre-requisite conflict

Further, we found that the pre-requisite relation is a DAG (Directed Acyclic Graph), because there's no double edge pre-requisite between two courses, and there's also no cyclic pre-requisite relationship between couple of courses. So that we could use topologic sort to find a possible order, then select substring from this order to avoid other constraints.

We first treat each course as a node, and the pre-requisite relationship between courses as an edge, and build a temporary table PreGraph to represent such link relationship.

```
1 DROP TEMPORARY TABLE IF EXISTS PreGraph;
2     CREATE TEMPORARY TABLE PreGraph AS
3     SELECT p.id_first AS prerequisite, p.id_second AS
          dependent
4     FROM Pre_requisite p
5     WHERE EXISTS (
6         SELECT 1 FROM TempCourses WHERE title = p.
              id_first OR title = p.id_second
7 );
```

Then, we counted the Indegree of all courses (that is, the number of pre-requisite courses currently available), all courses with current Indegree of 0 are added to the Queue, and topological sorting starts from these courses.

```
1 CREATE TEMPORARY TABLE InDegree AS
2     SELECT title, 0 AS degree
3     FROM TempCourses;
4
5     UPDATE InDegree
6     SET degree = (
```

```
7           SELECT COUNT(*)
8           FROM PreGraph
9           WHERE PreGraph.dependent = InDegree.title
10  );
11
12      CREATE TEMPORARY TABLE Queue AS
13      SELECT title
14      FROM InDegree
15  WHERE degree = 0;
```

Next, topological sorting is carried out, and the following operations are performed in the loop: Add a course from the Queue with Indegree 0 to the Sorted courses, update the Indegree of other adjacent courses on the graph, and then add the new courses with Indegree 0 to the Queue until all courses are selected.

```
1  SELECT COUNT(*) * 2 INTO max_iterations FROM
       TempCourses;
2           WHILE_LOOP: WHILE EXISTS (SELECT 1 FROM Queue)
                DO
3                   SET iteration_count = iteration_count +
                        1;
4                   IF iteration_count > max_iterations
                        THEN
5                       LEAVE WHILE_LOOP;
6                   END IF;
7
8                   SET @course = (SELECT title FROM Queue
                        LIMIT 1);
9                   DELETE FROM Queue WHERE title = @course
                        ;
10
11                  INSERT INTO SortedCourses
12                  SELECT * FROM TempCourses WHERE title =
                        @course;
13
14                  UPDATE InDegree
15                  SET degree = degree - 1
16                  WHERE title IN (SELECT dependent FROM
                        PreGraph WHERE prerequisite =
                        @course);
17
18                  CREATE TEMPORARY TABLE TempQueue AS
19                  SELECT title
20                  FROM InDegree
21                  WHERE degree = 0
```

```
                    AND title NOT IN (SELECT title FROM
                        Queue)
                    AND title NOT IN (SELECT title FROM
                        SortedCourses);

                INSERT INTO Queue
                SELECT title
                FROM TempQueue;

                DROP TEMPORARY TABLE TempQueue;
        END WHILE WHILE_LOOP;
```

3.2.3 Final Plan Decision With Other Constrains

Use cursor to go over courses after topological sort. Traverse the Sortedcourses, keep finding the next title that itself and its anti-requisite courses are not included in the plan. Among several courses of this title, find the earliest, unfilled session, which is before the students' graduation, and put it into the course selection scheme of that session. This part could also avoid the time conflict of the courses, which is similar to the method in short-term plan.

```
    DECLARE course_cursor CURSOR FOR
        SELECT title, course_id, semester, session_id,
            units
        FROM SortedCourses
        WHERE title NOT IN (SELECT DISTINCT title FROM
            Plan WHERE student_id = student_id);

    IF semester_part = 1 THEN
        SET cur_semester = 'Fall2024';
    ELSE
        SET cur_semester = 'Spring2025';
    END IF;
    SET max_year = 2024 + (5 - student_year);

    OPEN course_cursor;
    read_loop: LOOP
        FETCH course_cursor INTO cur_title,
            cur_course_id, cur_semester_cursor,
            cur_session_id_cursor, cur_units_cursor;
        IF done THEN
            LEAVE read_loop;
        END IF;
        SET max_year = 2024 + (5 - student_year);
        SET min_year = 2024;
```

```
21              IF cur_semester_cursor LIKE 'Fall%'
                    THEN
22          SET max_year = max_year - 1;
23      END IF;
24              IF (cur_semester_cursor LIKE 'Fall%')
                    AND (semester_part = 2) THEN
25          SET min_year = min_year + 1;
26      END IF;
27
28      IF NOT EXISTS (
29          SELECT 1
30          FROM Plan
31          WHERE title = cur_title
32      ) THEN
33          IF NOT EXISTS (
34              SELECT 1
35              FROM courses_taken
36              WHERE title = cur_title
37          ) THEN
38              IF NOT EXISTS (
39                  SELECT 1
40                  FROM Anti_requisite ar
41                  JOIN Plan p ON p.title = ar.id_2
42                  WHERE ar.id_1 = cur_title
43              ) THEN
44                  SELECT SUM(units) INTO
                        current_units
45                  FROM Plan
46                  WHERE student_id = student_id
47                    AND semester =
                          cur_semester_cursor
48                    AND session_id =
                          cur_session_id_cursor;
49
50                  IF (CAST(SUBSTRING(
                        cur_semester_cursor, -4) AS
                        UNSIGNED) <= max_year) AND (
                        CAST(SUBSTRING(
                        cur_semester_cursor, -4) AS
                        UNSIGNED) >= min_year) THEN
51                      IF current_units IS NULL OR
                            current_units +
                            cur_units_cursor <= 10 THEN
```

```
52                                  INSERT INTO Plan (
                                        student_id, course_id,
                                        title, semester,
                                        session_id, units)
53                                  VALUES (student_id,
                                        cur_course_id,
                                        cur_title,
                                        cur_semester_cursor,
                                        cur_session_id_cursor,
                                        cur_units_cursor);
54                                  SET current_units =
                                        current_units +
                                        cur_units_cursor;
55                              ELSE
56                                  ITERATE read_loop;
57                              END IF;
58                          END IF;
59                      ELSE
60                          ITERATE read_loop;
61                      END IF;
62                  END IF;
63              END IF;
64          END LOOP;
65          CLOSE course_cursor;
```

3.2.4 Overview of this function

The time complexity for topological sort will be $O(N + E)$, where N is the number of courses and E is the number of pre-requisite relations. The time complexity for final decision is Linear if we consider the number of anti-requisite is a small constant. Therefore, the total time complexity of making long term plan is also Linear to the Course number.

3.3 Time Conflict Check

Due to time constraints, the procedure for checking time conflicts was not applied in the final version of long-term course selection and the project demo.

3.3.1 Initial version of time conflict check:

```
1  CREATE PROCEDURE TimeConflict(student_id VARCHAR(20),
2   cur_semester_cursor VARCHAR(10), cur_session_id_cursor
3  VARCHAR(4))
4  BEGIN
5      DECLARE done INT DEFAULT 0;
```

```
6       DECLARE cur_course_id VARCHAR(20);
7       DECLARE cur_title VARCHAR(100);
8       DECLARE cur_start_time TIME;
9       DECLARE cur_end_time TIME;
10      DECLARE cur_units_cursor INT;
11      DECLARE time_conflict BOOLEAN DEFAULT FALSE;
12
13      DECLARE course_cursor CURSOR FOR
14          SELECT course_id, title, start_time, end_time,
                units
15          FROM SortedCourses
16          WHERE semester = cur_semester_cursor;
17
18      DECLARE CONTINUE HANDLER FOR NOT FOUND SET done =
            1;
19
20      OPEN course_cursor;
21
22      -- iterate over courses
23      read_loop: LOOP
24          FETCH course_cursor INTO cur_course_id,
25          cur_title, cur_start_time, cur_end_time,
26          cur_units_cursor;
27
28          IF done THEN
29              LEAVE read_loop;
30          END IF;
31
32          -- set time_conflict to false
33          SET time_conflict = FALSE;
34
35          -- check time conflict
36          IF EXISTS (
37              SELECT 1
38              FROM Plan
39              JOIN Courses ON Plan.course_id =
                    SortedCourses.course_id
40              WHERE Plan.student_id = student_id
41              AND Plan.semester = cur_semester_cursor
42              AND Plan.session_id = cur_session_id_cursor
43              AND (
44                  -- check if it's on the same day
45  ----------------(SortedCourses.days LIKE
46  ----------------CONCAT('%', SUBSTRING(
    cur_semester_cursor, 1, 2), '%')
```

```
47  ----------------OR SortedCourses.days LIKE
48  ----------------CONCAT('%', SUBSTRING(
        cur_semester_cursor, 3, 4), '%')
49  ----------------OR SortedCourses.days LIKE
50  ----------------CONCAT('%', SUBSTRING(
        cur_semester_cursor, 5, 6), '%')
51  ----------------OR SortedCourses.days LIKE
52  ----------------CONCAT('%', SUBSTRING(
        cur_semester_cursor, 7, 8), '%'))
53  ----------------AND
54  ------------------ check if the time slots overlap
55  ----------------NOT (SortedCourses.end_time <=
        cur_start_time OR
56  ----------------cur_end_time <= SortedCourses.
        start_time)
57  ------------)
58  --------) THEN
59  -------------- if there is a conflict, set
        time_conflict to TRUE
60  ------------SET time_conflict = TRUE;
61  --------END IF;
62
63  --------- insert course into Plan if there's no time
        conflict
64          IF time_conflict = FALSE
65          THEN
66              INSERT INTO Plan (student_id, course_id,
                    title, semester,
67              session_id, units)
68              VALUES (student_id, cur_course_id,
                    cur_title,
69              cur_semester_cursor, cur_session_id_cursor,
                     cur_units_cursor);
70          ELSE
71              -- skip the current course if there's time-
                    conflict
72  ------------ITERATE read_loop;
73  --------END IF;
74  ----END LOOP;
75  ----CLOSE course_cursor;
```

Code Explanation:
The cursor course_cursor is used to traverse the filtered courses. Each course is checked for time conflicts with courses already in table Plan which holds valid courses. This method will check if there is any course in Plan that meet the following conditions: (1) in the same semester, session, and day as the current course; (2) overlaps with

the time slot of the current course. The specific method for determining the time overlap is using NOT (SortedCourses.end_time = cur_start_time OR cur_end_time = SortedCourses.start_time) .

Time Complexity Analysis:
Worst case: The procedure will perform a time conflict check for each course. The worst-case scenario is that each check needs to compare the current course with all courses in the Plan, and the current course is added to the Plan after check. The first course requires 0 comparisons, and when N =1, the time complexity for checking the (N+1)th course is $O(N)$. Therefore, the worst-case time complexity for checking N courses is:
$O(0 + 1 + 2 + ... + (N - 1)) = O(N^2)$

Average-case Time Complexity: When checking the N-th course, number of courses in Plan is about N/2 on average. Therefore the average-case time complexity for checking N courses is
$O((0 + 1 + 2 + ... + (N - 1))/2) = O(N^2)$

As the number of courses increases, the time complexity of this method increases by the square, affecting performance, so a new method is adopted.

3.3.2 Optimized version of Time Conflict Check

First, we create table Schedule. It stores all possible class times from Monday to Thursday, from 8:30 to 20:30, with a 15- minute interval between every two time points. If the column occupied is 1, it means that the time point is occupied by a class; otherwise, the time point is not occupied.

```
CREATE TABLE Schedule (
    day_of_week VARCHAR(2),
    time_point TIME,
    occupied BOOLEAN DEFAULT 0,
    PRIMARY KEY (day_of_week, time_point)
);
```

Then, we create a procedure that can automatically insert values in Schedule and call it.

```
CREATE PROCEDURE GenerateSchedule()
BEGIN
    DECLARE start_time TIME DEFAULT '08:30:00';
    DECLARE end_time TIME DEFAULT '20:30:00';
    DECLARE current_time_point TIME;

    SET current_time_point = start_time;

    WHILE current_time_point <= end_time DO
        INSERT INTO Schedule (day_of_week, time_point)
```

```
11          VALUES ('Mo', current_time_point),('Tu',
                current_time_point),
12              ('We', current_time_point),('Th',
                    current_time_point);
13
14          SET current_time_point = ADDTIME(
                current_time_point, '00:15:00');
15      END WHILE;
16  END
```

Next, we create a procedure to check time conflict. The cursor course_cursor is used to get course information in SortedCourses.

```
1   CREATE PROCEDURE CheckTimeConflict()
2   BEGIN
3       DECLARE done INT DEFAULT 0;
4       DECLARE cur_course_id VARCHAR(20);
5       DECLARE cur_title VARCHAR(100);
6       DECLARE cur_semester VARCHAR(10);
7       DECLARE cur_session_id VARCHAR(4);
8       DECLARE cur_days VARCHAR(12);  -- e.g., MoTuWeTh
9       DECLARE cur_start_time TIME;
10      DECLARE cur_end_time TIME;
11      DECLARE current_time_point TIME;
12      DECLARE conflict BOOLEAN;
13
14      DECLARE course_cursor CURSOR FOR
15          SELECT course_id, title, semester, session_id,
                days,
16          start_time, end_time
17          FROM SortedCourses;
18
19      DECLARE CONTINUE HANDLER FOR NOT FOUND SET done =
            1;
```

Time Conflict Check Method:
Iterate over the sorted courses. For each course, start from the course's start time and check each time point until the end time. For the current time point, check the time point T that is the same as the current time point in table Schedule. If T is occupied by a class(column "occupied" in Schedule shows 1), then there's time conflict and the current course will be skipped.

```
1       read_loop: LOOP
2           FETCH course_cursor INTO cur_course_id,
                cur_title, cur_semester,
```

```
3           cur_session_id, cur_days, cur_start_time,
                cur_end_time;
4
5           IF done THEN
6               LEAVE read_loop;
7           END IF;
8
9           SET conflict = 0;
10              SET current_time_point = cur_start_time
                    ;
11
12          WHILE current_time_point < cur_end_time DO
13              IF FIND_IN_SET('Mo', cur_days) > 0 THEN
14                  IF EXISTS (
15                      SELECT 1
16                      FROM Schedule
17                      WHERE day_of_week = 'Mo'
18                      AND time_point = current_time_point
19                      AND occupied = 1
20                  ) THEN
21                      SET conflict = TRUE;
22                  END IF;
23              END IF;
24
25              IF FIND_IN_SET('Tu', cur_days) > 0 THEN
26                  IF EXISTS (
27                      SELECT 1
28                      FROM Schedule
29                      WHERE day_of_week = 'Tu'
30                      AND time_point = current_time_point
31                      AND occupied = 1
32                  ) THEN
33                      SET conflict = TRUE;
34                  END IF;
35              END IF;
36
37              IF FIND_IN_SET('We', cur_days) > 0 THEN
38                  IF EXISTS (
39                      SELECT 1
40                      FROM Schedule
41                      WHERE day_of_week = 'We'
42                      AND time_point = current_time_point
43                      AND occupied = 1
44                  ) THEN
45                      SET conflict = TRUE;
```

```
46                  END IF;
47              END IF;
48
49              IF FIND_IN_SET('Th', cur_days) > 0 THEN
50                  IF EXISTS (
51                      SELECT 1
52                      FROM Schedule
53                      WHERE day_of_week = 'Th'
54                      AND time_point = current_time_point
55                      AND occupied = 1
56                  ) THEN
57                      SET conflict = TRUE;
58                  END IF;
59              END IF;
60
61              IF conflict THEN
62                  ITERATE read_loop;
63              END IF;
64
65              -- If current time point is valid, check
                    next time point.
66              SET current_time_point = ADDTIME(
                    current_time_point, '00:15:00');
67          END WHILE;
```

If there's no time conflict, then insert the course to PlanA.
If the course is valid, update Schedule, set the time points in its time slot occupied.

```
1           SET current_time_point = cur_start_time;
2           WHILE current_time_point <= cur_end_time DO
3               IF FIND_IN_SET('Mo', cur_days) > 0 THEN
4                   UPDATE Schedule
5                   SET occupied = 1
6                   WHERE day_of_week = 'Mo' AND time_point
                        = current_time_point;
7               END IF;
8
9               IF FIND_IN_SET('Tu', cur_days) > 0 THEN
10                  UPDATE Schedule
11                  SET occupied = 1
12                  WHERE day_of_week = 'Tu' AND time_point
                        = current_time_point;
13              END IF;
14
15              IF FIND_IN_SET('We', cur_days) > 0 THEN
16                  UPDATE Schedule
```

```
17                      SET occupied = 1
18                      WHERE day_of_week = 'We' AND time_point
                            = current_time_point;
19                  END IF;
20
21                  IF FIND_IN_SET('Th', cur_days) > 0 THEN
22                      UPDATE Schedule
23                      SET occupied = 1
24                      WHERE day_of_week = 'Th' AND time_point
                            = current_time_point;
25                  END IF;
26
27                  SET current_time_point = ADDTIME(
                        current_time_point, '00:15:00');
28              END WHILE;
29          END LOOP;
30          CLOSE course_cursor;
31  END
```

Time Complexity Analysis:

The Schedule table has 48 time points. Each course's time slot has [5,20] time points and the worst-case time complexity of performing check at each time point = $O(48)$. Hence the complexity of iterating all the courses is $O(NC * 48) = O(N)$, N is number of courses, C is constant. Therefore, the time complexity of this method is $O(N)$. When there are a large number of courses to traverse, this method is better than the method in 4.3.1.

4 Simulation Demo

4.1 User Interface

This is the interface for user to input their personal information and query for course selection plan.

4.2 Information Filling

The information that can be collected in this interface includes:

1. **NetID**: A unique identifier for students to ensure the query is associated with the correct user.
2. **Class**: The student's current year of study
3. **Semester You Are In**: Allows students to specify the current semester, helping tailor recommendations to their current academic status.
4. **Major**: Students can input their declared major to receive course suggestions aligned with their field of study.

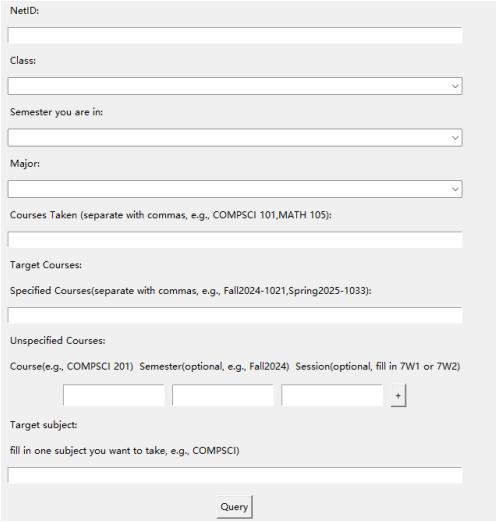

Fig. 3 Enter Caption

5. **Courses Taken**: A field to list previously completed courses, ensuring no duplicate recommendations and appropriate prerequisite checks.
6. **Target Courses**:
   - **Specified Courses**: Students can input specific courses they aim to take in the future. The ID for the courses can be found on DKUHUB.
   - **Unspecified Courses**: Students can provide general course information (e.g., course name and semester) to explore options.
7. **Target Subject**: An optional field where students can indicate a particular subject of interest to focus their course recommendations.
8. **Query Button**: Once all relevant fields are filled, students can click this button to generate their personalized course selection plan.

This user-friendly interface simplifies the course planning process, making it efficient and tailored to each student's needs.

This is an example for the information.

4.3 Query Result

The result includes both course selection for 4 years and for one semester.

If there is no valid course selection plan, the program will throw an erro, and the user can refill the information.

5 Conclusion

In order to solve the problem of course selection, this project has gone through the process of designing database, obtaining real data from DKUHub and bulletin files, designing specific course planner functions of short-term and long-term by using search

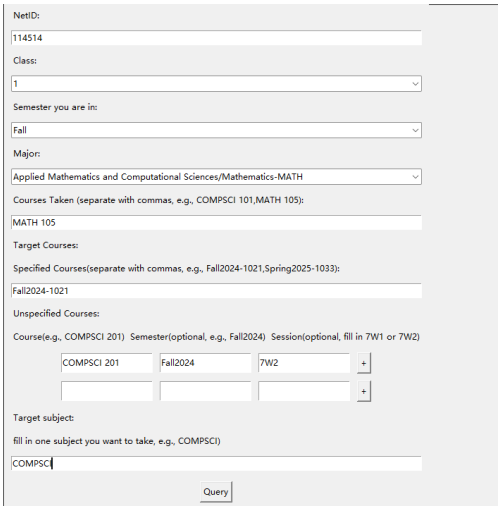

Fig. 4 Enter Caption

algorithm and topological sort, algorithm optimization, and designing user interaction. It basically solves the problem of short-term and long-term course selection and meets the individual needs of users.

Currently, we have only inserted the relevant course requirements for mathematics and computer science majors. We will improve the course requirements for other majors in the future. We also consider updating the course data in the database in real-time. If we get the user's permission, we can also conduct real-time prediction of the success rate of selection for each course based on the course selection of each user student with Logic regression in the future and give a course selection scheme with a higher success rate of selection. We could also recommend courses to users based on the course selection data of other users with similar choices.

6 Appendix

6.1 Contributions:

6.1.1 Database Schema

• Database schema version 1:
- Collective effort among Zi, Houmin, and Yuju
- Annotations written by Houmin, Yuju
• Database schema version 2:
– ER modeling designed by Houmin
– Changes to database schema reflected by Yuju
• Database schema final version(database_design.1.07.sql):
– Final changes to database schema by Zi

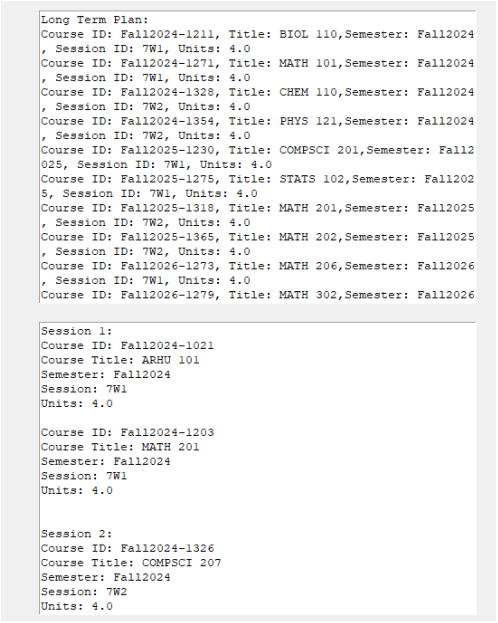

Fig. 5 Enter Caption

6.1.2 Database Data

• Data extract:
- Collective effort by Zi, Houmin
• Data insertion:
- Collective effort by Zi, Houmin

6.1.3 Algorithm

• Course selection for next semester(func1.py):
- Individual effort by Houmin
• Long-term course selection(longterm_select.sql):
- Individual effort by Zi
• Time conflict check for courses(CheckTimeConflict_v1.sql, CheckTimeConflict_Optimized.sql)
- Individual effort by Yuju

6.1.4 Interaction Design

• Front-end UI design:
– Individual effort by Yuju
• Front-end and back-end integration:
– Collective effort by Houmin, Yuju

6.1.5 Report

• Zi:
– Evaluaton of long-term course selection(4.2), Conclusion
• Houmin:
– Introduction, Evaluation of short-term course selection(4.1), Simulation Demo, Database Schema • Yuju:
– Abstract, Evaluation of time conflict check(4.3), Conclusion, Appendix

6.1.6 ReadMe file

- individual work by Houmin

6.1.7 Peer assessment

- Collective work by Zi, Houmin, Yuju

6.2 Timeline and Progress Notes

6.2.1 Week 1

The project proposal was made.

6.2.2 Week 2

The Software Requirements Specification (SRS) was made. The functionalities of the product, UI input/output, and basic system design were outlined.

6.2.3 Week 3

The first version of database design was made. The database schema was visualized.

6.2.4 Week 4

The ER model was designed.
Changes to database schema:
1. Some tables need to reference "title" in the "Courses" table, but "title" was not the primary key. Hence we split "Courses" table into two tables: "Courses" and "Course_info". In "Courses_info", "title" is the primary key.
2. "course_id" in tables storing prerequisite and anti-requisite relationships of courses was changed to "title". This can avoid same title in these tables being stored repeatedly (since many "course_id" can correspond to one "title").

6.2.5 Week 5

Group work began on database data extract, data insertion, and UI design. For data extract, we used Selenium to log into DKUHUB. However, due to privacy policy on the website, we were unable to use Selenium to get personal information of users such as courses students had taken. Hence we decided to ask students to manually input the courses they had taken when using the product.

6.2.6 Week 6

We began designing algorithms for course selection for the next semester and long-term course selection. UI design continued.

6.2.7 Week 7

Frontend and backend integration was made. We made some changes to the parameters in the UI to better fit the course selection function and procedure. The algorithm for checking time conflicts of courses was designed in SQL. To optimize the short-term course selection, the topology sorting algorithm was applied. Project report written.