MATH535 Model

Prerequisite packages

In [1]:
```
!pip install cplex
!pip install docplex
import sys
from docplex.cp.model import *
import pandas as pd
import random
```

```
Requirement already satisfied: cplex in /Users/alishakhan/opt/miniconda3/envs/alishadev/lib/python3.9/site-pack
ages (22.1.0.0)
Requirement already satisfied: docplex in /Users/alishakhan/opt/miniconda3/envs/alishadev/lib/python3.9/site-pa
ckages (2.23.222)
Requirement already satisfied: six in /Users/alishakhan/opt/miniconda3/envs/alishadev/lib/python3.9/site-packag
es (from docplex) (1.15.0)
```

Getting data for our model

Importing CSV file of the courses from IIT course status report. We only keep columns we need, so we're dropping the rest.

In [2]:
```
#Every course section in IIT fall 2022 semester, from IIT course status report
df = pd.read_csv("CourseStatusReport--2022-04-16.csv")
df = df.drop(columns=["Status", "Cross List Code", "College", "Department", "Schedule Type", "Max Enrollment",
```

We don't have a way of getting course rating and professor difficulty, so we populate this data with random numbers for this project. We maintain a consistent seed to maintain consistent results, though.

In [3]:
```
random.seed(2022)
df["Course Quality"]=[random.randrange(5) for i in range(len(df))]
df["Professor Difficulty"]=[random.randrange(5) for i in range(len(df))]
```

We also have a seperate ID column. A CRN might work, but many lab and recitation sections of a course have the same CRN but are listed as 2 sperate rows, so we need a seperate ID number to be a unique identifier

In [4]:
```
df["ID"] = range(len(df))
```

Creating the model

This is our model

In [5]:
```
model=CpoModel()
```

Here is where we create the variables for our models. Each course section is represented with a binary variable, that is, an integer variable restricted to only the values 0 and 1. Linear optimization variables are represented in cplex just as any other programming variable, so we have to store all of the class variables in the array `classvariables`.
We also add the cplex variable associated with each course section to the dataframe row corresponding that course

In [6]:
```
#creating binary vars for each class
classVariables=[]
for index, row in df.iterrows():
    classVariables.append(model.integer_var(min=0, max=1, name="id"+str(row['ID'])))
    #print("crn"+str(row['CRN']))
df['Vars']=classVariables
```

Adding Constraints

The first constraint we add is course section. We want to ensure that our model does not suggest to take 2 sections of the same course. To do this, we first create a dictionary. The keys are the course number (for example "MATH 535") and the values are a list of course variables corresponding to a section of the course. We would make the sum of all variables in this list be either 1, 0, or either, depending on whether this course is required, requested, or neither, respectively. This information is found out later, so we make the constraints themselves then.

In [7]:
```
#Course sections constraint
#Make a dictionary. Keys: Course subject course number touple. Values: variables
sections2 = {}
for idx, x in enumerate(df["Course Code"]): #List through all Course Codes. Consider the place before the hyphe
    #print(x.split("-")[0])
    x = x.split("-")[0]
    if(x not in sections2):
        sections2[x] = [classVariables[idx]]
    else:
        sections2[x].append(classVariables[idx])
#For each key in the dictionary, add a constraint
# for coursecode in sections2:
#     #For each constraint: the sum of all CRN's is equal to 1
#     allsectionssum = model.sum(1*(sections2[coursecode][i]) for i in range(len(sections2[coursecode])))
#     model.add_constraint(allsectionssum <= 1)
#     #print(allsectionssum)
```

The next constraint we add is time slot. The time slot constraint is that the sum of all courses taken at the same time slot must be 1. So the first thing we do is we create a dictionary. The keys are the time slots, and the values are lists of course variables corresponding to that time slot. We immediately create the constraint that the sum of the number of all courses in the same time slot is at most 1. In other words, there either is 1 course at a certain time slot, or there are none.

In [8]:
```
#Course time slot constraints
#Make a dictionary. Keys: Course time slot. Values: ID's
timeslots = {}
for idx, x in enumerate(df["Time"]): #List through all Time slots"
    if((type(df["Days"][idx]))!=float):
        for char in (df["Days"][idx]):
            timeday = char+x
            if(timeday not in timeslots):
                timeslots[timeday] = [classVariables[idx]]
            else:
                timeslots[timeday].append(classVariables[idx])
#For each key in the dictionary, add a constraint
for timeslot in timeslots:
    #For each constraint: the sum of all CRN's is equal to 1
    alltimeslotssum = model.sum(1*(timeslots[timeslot][i]) for i in range(len(timeslots[timeslot])))
    model.add_constraint(alltimeslotssum <= 1)
```

In [9]:
```
# #Credit count: The sum of the credit count for all taken sections is less that 18 and bigger than 12
# creditCount = [0]*len(df)
# for i in range(len(df)):
#     creditCount[i]=df["Credits"][i]
# creditCount[:5]
```

Here we ask the user for the minimum and maximum credit count allowed for the schedule. These feed directly into a constraint: the sum of the credit value of all courses selected by the schedule cannot exceed the maximum credit count inputted and cannot be less than the minimum credit count inputted.

In [10]:
```
required_classes=[]
mincredits = int(input("Enter a min credits you need to take "))
maxcredits = int(input("Enter a max credits you need to take "))
```

```
Enter a min credits you need to take 12
Enter a max credits you need to take 18
```

In [11]:
```
allcreditcountsum = model.sum(df["Credits"][i] * df["Vars"][i] for i in range(len(df)))
model.add_constraint(allcreditcountsum <= maxcredits)
model.add_constraint(allcreditcountsum >= mincredits)
```

In [12]:
```
#observe that constraints were added
#model.print_information()
```

Here is where we ask for the specific classes required and requested.
Required courses are courses that must be a part of the schedule. Requested courses are courses that could be a part of the schedule. We assume that courses listed as neither required or requested courses are not to be taken at all.

As an example, we give a situation with many possible choices: a first semester CS major. A CS major must take CS100 and CS201 as soon as possible, but there are many other courses also availible to take in the meantime which are listed in the input for cell for requested classes.

In [16]:
```
required_classes=[]
val="class"
val=input("Enter a course you need to take ")
while val.lower()!="exit":
    required_classes.append(val)
    val=input("Enter another course you need to take. \nType \"Exit\" to stop adding courses ")
```

```
Enter a course you need to take CS 100
Enter another course you need to take.
Type "Exit" to stop adding courses CS 201
Enter another course you need to take.
Type "Exit" to stop adding courses EXIT
```

In [17]:
```
requested_classes=[]
val="class"
val=input("Enter a course you want to take ")
while val.lower()!="exit":
    requested_classes.append(val)
    val=input("Enter another course you want to take. \nType \"Exit\" to stop adding courses ")
```

```
Enter a course you want to take MATH 151
Enter another course you want to take.
Type "Exit" to stop adding courses CHEM 124
Enter another course you want to take.
Type "Exit" to stop adding courses PHYS 123
Enter another course you want to take.
Type "Exit" to stop adding courses CS 350
Enter another course you want to take.
Type "Exit" to stop adding courses PSYC 303
Enter another course you want to take.
Type "Exit" to stop adding courses EXIT
```

In [ ]:
```
#Seeing if coursecode keys are strings
#for coursecode in sections2:
 #   print(coursecode, " ", type(coursecode))
```

Here we finish implementing the course section constraint. Note that for required classes, there must be exactly one section taken, and for requested classes, it is up to 1. For all other classes, this sum must be 0 (ie we ignore classes that are neither required nor requested).

In [18]:
```
#Adding constraints for what courses to take, and identical course conflict avoidance
for coursecode in sections2:
    #For each constraint: the sum of all CRN's is equal to 1
    allsectionssum = model.sum(1*(sections2[coursecode][i]) for i in range(len(sections2[coursecode])))
    if(coursecode in required_classes):
        model.add_constraint(allsectionssum == 1)
    elif(coursecode in requested_classes):
        model.add_constraint(allsectionssum <= 1)
    else:
        model.add_constraint(allsectionssum == 0)
```

Adding cost function

Here is where we construct our cost funtion ratios, as described earlier in our report. We ask thse questons here

In [19]:
```
a = int(input("""Hypothetically, there's a 1/5 quality course at a non-8:35 AM time slot. How good does the cou

b = int(input("""For every 1 point increase in professor difficulty, how many points better should their course
```

```
Hypothetically, there's a 1/5 quality course at a non-8:35 AM time slot. How good does the course have to be (1
-5) to justify moving to 8:35 AM?  1
For every 1 point increase in professor difficulty, how many points better should their course be to justify i
t? 1
```

Here we generate the function to minimize. So every positive term we add is a "penalty" for a schedule. There is a penalty for professor difficulty, and a penalty for course quality's distance from 5 (the highest score). We add this penalty to the minimization function. This penalty is applied for every course section.

In [20]:
```
#creating minimization function (course quality and prof difficulty)
min_func=0
for index, row in df.iterrows():
    min_func+=(((5-row['Course Quality']))+(a*row['Professor Difficulty']))*row['Vars']
```

We also add a penalty (whose magnitude depends on user input) for having a class be at 8:35am. This decision is arbitrary, but it's there to demonstrate that this model can incorperate many properties about a course section into the cost function.

In [21]:
```
earlyClassPenalty = (b-1)
for index, row in df.iterrows():
    if(row["Time"]=="0835 - 0950"):
        min_func += (earlyClassPenalty * row['Vars'])
```

Here we finalize the minimization problem and add it to our model

In [22]:
```
min_func=model.minimize(min_func)
model.add(min_func)
```

In [23]:
```
#observe that constraints were added
#model.print_information()
```

Solving the model

In [24]:
```
#solving model
sol2=model.solve()
#sol2.print_solution()
```

```
 ! --------------------------------------------------- CP Optimizer 22.1.0.0 --
 ! Minimization problem - 2619 variables, 1114 constraints
 ! Presolve      : 40 extractables eliminated
 ! Initial process time : 0.07s (0.07s extraction + 0.00s propagation)
 !  . Log search space  : 70.0 (before), 70.0 (after)
 !  . Memory usage      : 3.9 MB (before), 3.9 MB (after)
 ! Using parallel search with 8 workers.
 ! ----------------------------------------------------------------------------
 !          Best Branches  Non-fixed    W       Branch decision
                        0         70                 -
 + New bound is 0
                        0         70    1   F        -
 + New bound is 4
                       28         16    1   F     0 != id593
 + New bound is 5
 *          23        91  0.15s         1      (gap is 78.26%)
 *          20       242  0.15s         1      (gap is 75.00%)
 *          16       342  0.16s         1      (gap is 68.75%)
 *          15       729  0.16s         1      (gap is 66.67%)
 *          12       827  0.16s         1      (gap is 58.33%)
            12      1000          1     1        1 != id787
            12      1000          1     2        0  = id1772
 *          10       930  0.16s         3      (gap is 50.00%)
 *           9       973  0.16s         3      (gap is 44.44%)
             9      1000          1     3          -
             9      1000          1     4        1 != id743
             9      1000          1     5   F    0 != id1764
             9      1000          1     6   F    0 != id1765
             9      1000          1     7   F    1  = id741
             9      1000          2     8   F    0  = id2428
             9      1463          6     7          -
 + New bound is 9 (gap is 0.00%)
 ! ----------------------------------------------------------------------------
 ! Search completed, 7 solutions found.
 ! Best objective         : 9 (optimal - effective tol. is 0)
 ! Best bound             : 9
 ! ----------------------------------------------------------------------------
 ! Number of branches     : 14082
 ! Number of fails        : 5866
 ! Total memory usage     : 46.2 MB (45.6 MB CP Optimizer + 0.6 MB Concert)
 ! Time spent in solve    : 0.21s (0.14s engine + 0.07s extraction)
 ! Search speed (br. / s) : 100585.8
 ! ----------------------------------------------------------------------------
```

Here we show an optimal solution to the model. It generally makes sense, as it looks like a typical schedule of a first semester CS student

We note that it chose a virtual section of CS 201, which allowed for more flexibility with other class's time slots. This makes sense, as several students use exactly this strategy

In [25]:
```
print("Here's your \"optimal\" schedule: ")
for index, row in df.iterrows():
    #print(sol2.get_value("id"+str(index)))
    if(sol2.get_value("id"+str(index))!=0):
        print(df["Course Code"][index], "on", df["Days"][index],"at", df["Time"][index])
```

```
Here's your "optimal" schedule:
CS 100-L05 on T at 1350 - 1505
CS 201-03 on nan at  -
MATH 151-01 on MWF at 0835 - 0950
PSYC 303-01 on MW at 1125 - 1240
```

And here we show all feasible solutions, which includes solutions that are not quite as optimal but present alternatives in case what the model outputted is either undesirable for reasons beyond what the model tracks or a user just wants to see what other options there might be. The last solution is the optimal one.

In [26]:
```
lsols=model.start_search(SearchType='DepthFirst', Workers=1)
index
for sol in lsols:
    #sol.write()
    print("\nHere is a feasible solution")
    for index, row in df.iterrows():
        if(sol.get_value("id"+str(index))!=0):
            print(df["Course Code"][index], "on", df["Days"][index],"at", df["Time"][index])
```

```
 ! --------------------------------------------------- CP Optimizer 22.1.0.0 --
 ! Minimization problem - 2619 variables, 1114 constraints
 ! Presolve      : 40 extractables eliminated
 ! Workers               = 1
 ! SearchType            = DepthFirst
 ! Initial process time : 0.04s (0.04s extraction + 0.00s propagation)
 !  . Log search space  : 70.0 (before), 70.0 (after)
 !  . Memory usage      : 3.9 MB (before), 3.9 MB (after)
 ! Using sequential search.
 ! ----------------------------------------------------------------------------
 !          Best Branches  Non-fixed            Branch decision
 *            15      271  0.04s

Here is a feasible solution
CHEM 124-03 on TR at 1125 - 1240
CS 100-L04 on M at 1825 - 1940
CS 201-02 on TR at 1350 - 1505
MATH 151-01 on MWF at 0835 - 0950
 *            14      276  0.04s

Here is a feasible solution
CHEM 124-03 on TR at 1125 - 1240
CS 100-L04 on M at 1825 - 1940
CS 201-03 on nan at  -
MATH 151-01 on MWF at 0835 - 0950
 *            10      279  0.04s

Here is a feasible solution
CHEM 124-03 on TR at 1125 - 1240
CS 100-L12 on W at 1825 - 1940
CS 201-03 on nan at  -
MATH 151-01 on MWF at 0835 - 0950
              10     1000          1              0  = id748
 *             9     1280  0.05s

Here is a feasible solution
CHEM 124-01 on MW at 1000 - 1115
CS 100-L05 on T at 1350 - 1505
CS 201-03 on nan at  -
MATH 151-05 on MWF at 1125 - 1240
 ! ----------------------------------------------------------------------------
 ! Search completed, 4 solutions found.
 ! Best objective         : 9 (optimal - effective tol. is 0)
 ! Best bound             : -Infinity (no gap)
 ! ----------------------------------------------------------------------------
 ! Number of branches     : 1386
 ! Number of fails        : 690
 ! Total memory usage     : 12.8 MB (12.2 MB CP Optimizer + 0.6 MB Concert)
 ! Time spent in solve    : 0.06s (0.02s engine + 0.04s extraction)
 ! Search speed (br. / s) : 69300.1
 ! ----------------------------------------------------------------------------
```

In [ ]: