Tournament Scheduling

In this notebook, we will go through how the Wenatchee AppleSox clinic team implemented tournament schedule generation.

We'll start by importing the class and installing the necessary preliminaries. Please upload sportsSolver.py and Scheduler.py in the files section on the left panel of Google Colaboratory, or make sure these files are in the same directory as this notebook. You can find these files on our github at https://github.com/HMCAppleSox/scheduleit/tree/master/scheduleit/api.

```
!pip install mip
```

```
from sportsSolver import SportsSolver
```

The Scheduling Class

The constructor

Currently, when we instantiate a solver object, it takes in schedule constraints as arugments.

Variable Name	Description	Required?	Default
`teams`	a list of teams that are playing	Yes	
`wpw_matrix`	a "who plays who" dictionary that encodes how many times each team plays another home and away	Yes	
`cost_matrix`	a cost dictionary that encodes the cost of travel between each team	No	`None`
`desired_games`	slots where teams wish to play at home	No	`[]`
`blackout_games`	slots where teams wish to play away	No	`[]`
`starts_on_weekend`	whether or not the first slot of the schedule starts on a weekend	No	`True`
`min_weekends_at_home`	the minimum number of weekend slots any team is allowed to have	No	`0`
`max_cons_home_games`	the maximum number of consecutive games where teams play at home	No	`0`
`max_cons_away_games`	the maximum number of consecutive games where teams play away	No	`0`
`start_date`	the earliest date in which the schedule can start	No	`None`
`should_shuffle`	a boolean marking whether the input teams should be shuffled	No	`True`
`verbose`	a boolean indicating whether to print helpful info to the terminal	No	`True`

We used the open source library MIP to implement integer linear programming. Each sportsSolver object is initialized with an MIP Model to which we add integer constraints and then solve. The model uses the instance variable `plays` to hold all the possible games that could be played -- every possible combination of home team, away team, and slot (date). Then, to add constraints, we simply enforce some aspect of `plays` . The methods that add constraints are called in the constructor itself. Let's go into more detail about how that works.

Adding constraints to the model

For example, to ensure that the who plays who constraint is satisfied, we iterate through every possible matchup of teams and ensure that the number of times that match appears in `plays` is at least the number of times indicated in the who plays who input constraint.

```
 def add_who_plays_who(self):
        self.added_constraints.append('who plays who')
        for team1 in self.teams:
            for team2 in self.teams:
                num_games_to_play_at_team2 = self.wpw_matrix[(team1, team2)]
                num_games_played_at_team2 = [self.plays[(match, slot)] for match in self.matches for slot in
                self.model.add_constr(xsum(num_games_played_at_team2) >= num_games_to_play_at_team2)
```

Solving

In this section, we'll go through an examples to understand how to run the code.

Small example

We'll begin by defining all of our variables. We'll do a small example with only 8 teams, where each team plays each other team once and the cost of travel between teams is constant. Let's define all the variables that are needed for the constructor.

```
teams = ["WEN", "YAK", "NAN", "WWS", "COW", "BEL", "KEL", "VIC"]
wpw_matrix = {}
for t1 in teams:
    for t2 in teams:
        if t1 != t2:
            wpw_matrix[(t1, t2)] = 1
        else:
            wpw_matrix[(t1, t2)] = 0
cost_matrix = {}
for t1 in teams:
    for t2 in teams:
        if t1 != t2:
            wpw_matrix[(t1, t2)] = 1
        else:
            wpw_matrix[(t1, t2)] = 0
desired_games = [("WEN", 1), ("YAK", 2), ("COW", 1)]
blackout_games =  [("KEL", 1), ("WWS", 1),("NAN", 2), ("VIC", 2)]
starts_on_weekend = True
min_weekends_at_home = 3
max_cons_home_games = 4
max_cons_away_games = 6
start_date = "11/03/2021"
```

Now, we build the solver instance and find a solution using `solve()`:

```
small_solver = SportsSolver(teams,
  wpw_matrix,
  cost_matrix=cost_matrix,
  desired_games=desired_games,
  blackout_games=blackout_games,
  starts_on_weekend=starts_on_weekend,
  min_weekends_at_home=min_weekends_at_home,
  max_cons_home_games=max_cons_home_games,
  max_cons_away_games=max_cons_away_games)
```

```
small_solver.solve()
```

```
True
```

Let's print out the result. The printout is explained in the next section.

```
print(small_solver)
```

```
====== slot 0 ======
NAN plays KEL at NAN
COW plays BEL at COW
WEN plays VIC at WEN
YAK plays WWS at YAK

====== slot 1 ======
WWS plays VIC at WWS
KEL plays COW at KEL
YAK plays NAN at YAK
BEL plays WEN at BEL

====== slot 2 ======
WWS plays COW at WWS
VIC plays YAK at VIC
WEN plays NAN at WEN
BEL plays KEL at BEL

====== slot 3 ======
NAN plays VIC at NAN
COW plays WEN at COW
YAK plays KEL at YAK
BEL plays WWS at BEL

====== slot 4 ======
WWS plays NAN at WWS
KEL plays WEN at KEL
YAK plays VIC at YAK
BEL plays COW at BEL

====== slot 5 ======
WWS plays BEL at WWS
NAN plays YAK at NAN
WEN plays COW at WEN
KEL plays VIC at KEL

====== slot 6 ======
WWS plays WEN at WWS
NAN plays COW at NAN
VIC plays KEL at VIC
BEL plays YAK at BEL

====== slot 7 ======
COW plays WWS at COW
VIC plays NAN at VIC
WEN plays YAK at WEN
KEL plays BEL at KEL

====== slot 8 ======
NAN plays WEN at NAN
COW plays VIC at COW
KEL plays WWS at KEL
YAK plays BEL at YAK

====== slot 9 ======
WWS plays YAK at WWS
```

```
    VIC plays COW at VIC
    WEN plays BEL at WEN
    KEL plays NAN at KEL
```

Understading the Output

The output is divided into three sections:

1. the schedule
2. the travel costs
3. the constraints

The Schedule

Let's first begin to understand the schedule section by looking at a section of a sample output:

```
.
.
.
====== slot 4 ======
YAK plays COW at YAK
WWS plays WEN at WWS
BEL plays KEL at BEL
VIC plays NAN at VIC

====== slot 5 ======
WEN plays WWS at WEN
VIC plays BEL at VIC
KEL plays NAN at KEL
COW plays YAK at COW

====== slot 6 ======
WEN plays BEL at WEN
YAK plays VIC at YAK
KEL plays WWS at KEL
COW plays NAN at COW
.
.
.
```

The schedule is divided into slots. Depending on how your tournament is being run, each slot could span a certain number of games. For example, the West Coast League (the collegiate summer baseball league which is home to the Wenatchee Applesox) has each slot spanning 3 days, where each match is played every day during that slot.

Each slot contains a list of the occuring matches in plain english. For example:

```
YAK plays COW at YAK
```

Means that YAK will be playing COW during that slot at YAK's stadium.

The Travel Costs

Beneath the last slot displayed in the output, there will be a section titled `Travel Costs:`. Let's take a look at an example.

```
Travel Costs:
KEL: 5719
NAN: 4444
YAK: 3646
WEN: 3398
VIC: 3386
COW: 3805
BEL: 2813
WWS: 3959
---
average: 3896.25
```

The total cost of travel is calculated with values from the cost matrix that the user has inputted. We then display the average travel cost of all the teams as a metric for determining how costly a particular schedule is.

The Constraints

Beneath the travel costs section there will be a list of applied constraints. Let's take a look at an example.

```
Constraints Applied:
- each team plays once
- max roadtrip
- max homestay
- who plays who
```

Here, we see that 4 constraints are being enforced:

- each team plays once during a slot
- a maximum roadtrip length
- a maximum homestay length
- the who plays who matrix

This list of applied constraints will be different depending on what constraints are added to the solver.

Excel Output

In addition to the terminal output, you are also able have the schedule exported to an excel spreadsheet. This can be done via the command line interface using the `-f` option, or simply by calling the `write_to_excel` function from `Scheduler.py`.