Switching from CPLEX to MIP: The Journey

This semester, the Wenatchee Applesox clinic team started with migrating the backend code from using CPLEX to instead use MIP, mixed integer linear programming. The main reasons as to why this decision was made was due to costs. As we were preparing for deployment, we ran into the problem of increased costs in terms of using CPLEX. The numbers were beyond our budget, and so we decided to explore different options. During this process, we looked into using Gurobi or Knuth but both had issues related to cost or scaling. Thus, we decided to move forward with MIP. What sets MIP apart from these other options is that it is open sourced, which comes with many different benefits. Our biggest concern of cost was no longer a worry with MIP. More importantly, open source programs allow for public collaboration such that they are improved continuously over time and are reliable due to large user bases. With that being said, MIP specifically met our needs due to its fast multi solving nature which was perfect for solving our sports scheduling problem. After making this decision, we spent the following weeks performing the migration.

Luckily, the migration process was more straightforward than expected. We changed the pre-existing solver code to be object oriented and translated our constraints to fit MIP. Let us take a look at an example of a constraint that was changed in this process. Figure 1 is a snippet of one of the constraints we had for the CPLEX solver. This constraint ensures that each team only plays once and there are no duplicates.

```
def each_team_plays_once(mdl,slots):
    for s in slots:
        for t in mdl.team_range[:-1]:
            max_teams_in_division = (mdl.plays[m, s] for m in mdl.matches if m.team1 == t or m.team2 == t)
            mdl.add_constraint(mdl.sum(max_teams_in_division) == 1,
                               "plays_exactly_once_%d_%s" % (s, t))
```

(Figure 1: CPLEX Compatible “each_team_plays_once” Constraint)

Then, in comparison, we have Figure 2, which is the new code that we have written this semester. The general structure of using the nested for loops to traverse through the slots is retained in this code. However, there are a few differences as well. First, we see that the MIP version of the code no longer takes in external parameters because *self* contains all of the needed information with object oriented code. Second, we utilize BYEs in this code to check whether a team is playing. Overall, due to the generally similar structure, the code translation process went smoothly.

```
def add_each_team_plays_once(self):
    for game_slot in range(1, self.num_slots + 1):
        for team in self.teams:
            if team != 'BYE':
                plays_in_slot = [self.plays[(match, slot)] for (match, slot) in self.plays if slot == game_slot and (match.home_team == team or match.away_team == team)]
                self.model.add_constr(xsum(plays_in_slot) == 1)
```

(Figure 2: MIP Compatible “add_each_team_plays_once” Constraint)

All of the translated code now resides in Scheduler.py with comments that detail each function and the general code itself. A lot of the translation process was aided by public resources from MIP and CPLEX. We now have a sport schedule solver that runs purely using MIP.