Sports scheduling

Sports scheduling

How can a sports league schedule games between teams in different divisions such that the teams play each other the appropriate number of times and maximize the objective of scheduling intradivision matches as late as possible in the season?

A sports league with two divisions needs to schedule games such that each team plays every team within its division a given number of times and plays every team in the other division a given number of times. Each week, a team plays exactly one game. A pair of teams cannot play each other on consecutive weeks. While a third of a team's intradivisional games must be played in the first half of the season, the preference is for intradivisional games to be held as late as possible in the season. To model this preference, there is an incentive for intradivisional games; this incentive increases each week as a square of the week. The problem consists of assigning an opponent to each team each week in order to maximize the total of the incentives.

This type of discrete optimization problem can be solved using Integer Programming (IP) or Constraint Programming (CP). Integer Programming is the class of problems defined as the optimization of a linear function, subject to linear constraints over integer variables. Constraint Programming problems generally have discrete decision variables, but the constraints can be logical, and the arithmetic expressions are not restricted to being linear. This example uses IP; the file `Sport-scheduling.pdf` presents the same problem using CP.

The data (`Sport-scheduling.dat`)

For DropSolve, all data must be defined in the form of tables (which in OPL implies using sets of tuples to define these tables).

In this sports league problem, the data is simple. There are eight teams in each division, and the teams must play each team in the division once and each team outside the division once.

```
scheduleParams =
<
   8, // nb teams in division
  16, // max teams in division
   1, // number of matches to play inside division.
   1  // number of matches to play outside division.
>;
```

Also given in the data is information regarding the 32 possible teams in the league, 16 in each division.

```
teamDiv1 =
{
   <"Baltimore Ravens">
 , <"Cincinnati Bengals">
 , <"Cleveland Browns">
 , <"Pittsburgh Steelers">
 , <"Houston Texans">
 , <"Indianapolis Colts">
 , <"Jacksonville Jaguars">
 , <"Tennessee Titans">
 , <"Buffalo Bills">
 , <"Miami Dolphins">
```

```
 , <"New England Patriots">
 , <"New York Jets">
 , <"Denver Broncos">
 , <"Kansas City Chiefs">
 , <"Oakland Raiders">
 , <"San Diego Chargers">
};

teamDiv2 =
{
    <"Chicago Bears">
  , <"Detroit Lions">
  , <"Green Bay Packers">
  , <"Minnesota Vikings">
  , <"Atlanta Falcons">
  , <"Carolina Panthers">
  , <"New Orleans Saints">
  , <"Tampa Bay Buccaneers">
  , <"Dallas Cowboys">
  , <"New York Giants">
  , <"Philadelphia Eagles">
  , <"Washington Redskins">
  , <"Arizona Cardinals">
  , <"San Francisco 49ers">
  , <"Seattle Seahawks">
  , <"St. Louis Rams">
};
```

The model (Sport-schedCPLEX.mod)

Given the number of teams in each division and the number of intradivisional and interdivisional games to be played, it is possible to calculate the total number of teams and the number of weeks in the schedule, assuming every team plays exactly one game per week. The season is split into halves, and the number of the intradivisional games that each team must play in the first half of the season is calculated.

```
/// There are two divisions.
int nbTeams = 2 * nbTeamsInDivision;
range Teams = 1..nbTeams;

/// Calculate the number of weeks necessary.
int nbWeeks = (nbTeamsInDivision-1)* nbIntraDivisional
              + nbTeamsInDivision * nbInterDivisional;
range Weeks = 1..nbWeeks;

/// Season is split into two halves
range FirstHalfWeeks = 1..ftoi(floor(nbWeeks/2));
int nbFirstHalfGames = ftoi(floor(nbWeeks/3));
```

For each pair of teams, a tuple is created. The tuple includes the index of each team and an integer which indicates whether or not the two teams are in the same division. The set of all tuples is called `Matches`. Additionally, an array of integers called `nbPlay` is created over the tupleset of pairs to model the number of games that each pair must play.

```
/// All possible matches (pairings) and whether or not each is intradivisional.
tuple Match {
  int team1;
  int team2;
  int isDivisional;
};
{Match} Matches = {<t1,t2,
                    (( t2 <= nbTeamsInDivision || t1 > nbTeamsInDivision) ? 1 : 0)>
                   | t1,t2 in Teams : t1<t2};
```

```
/// Number of games to play between pairs depends on
/// whether the pairing is intradivisional or not.
int nbPlay[m in Matches] = m.isDivisional==1 ? nbIntraDivisional : nbInterDivisional;
```

Modeling the variables and expressions

A solution to the problem can be modeled by determining whether or not a pair of teams plays in a particular week. The array of binary decision variables `plays` is indexed over the tupleset `Matches` and the `Weeks`. The value of the decision variable `plays[m][w]` indicates whether or not the pair of teams in tuple `m` plays in week `w`.

```
// Boolean decision variables : 1 if match m is played in week w, 0 otherwise.
dvar boolean plays[Matches][Weeks];
```

Modeling the constraints

One set of constraints is used to ensure the solution satisfies the number of intradivisional and interdivisional games that each team must play. The number of games a pair `m` is required to play over all the weeks is represented by `nbPlay[m]`.

```
// Each pair of teams must play the correct number of games.
forall (m in Matches)
  correctNumberOfGames:
  sum(w in Weeks) plays[m][w] == nbPlay[m];
```

Each week, every team must be assigned to exactly one game. For a given week and team, the sum of all the games the team plays must be 1.

```
// Each team must play exactly once in a week.
forall (w in Weeks, t in Teams)
  playsExactlyOnce:
  sum(m in Matches : (m.team1 == t || m.team2 == t)) plays[m][w] == 1;
```

A pair of teams cannot play each other on consecutive weeks.

```
// Games between the same teams cannot be on successive weeks.
forall (w in Weeks, m in Matches)
  cannotPlayAgain:
  if ( w < nbWeeks ) plays[m][w] + plays[m][w+1] <= 1;
```

Each team must play at least a certain number of intradivisional games, `nbFirstHalfGames`, in the first half of the season.

```
// Some intradivisional games should be in the first half.
forall (t in Teams)
  inDivisionFirstHalf:
  sum(w in FirstHalfWeeks, m in Matches : (((m.team1 == t || m.team2 == t)
                                           && m.isDivisional == 1 )))
            plays[m][w] >= nbFirstHalfGames;
```

Modeling the objective

The objective function for this example is designed to force intradivisional games to occur as late in the season as possible. The incentive for intradivision games increases by week. There is no incentive for interdivisional games. The incentive, or gain, is a power function of the week.

```
/// The goal is for intradivisional games to be played late in the schedule.
/// Only intradivisional pairings contribute to the overall gain.
int Gain[w in Weeks] = w * w;
```

The objective is to maximize the gain. If a pair of teams `m` plays a game in week `w` and the pair is an intradivisional match, the value `Gain[w]` is added to the objective value.

```
// If an intradivisional pair plays in week w, Gain[w] is added to the objective.
maximize sum (m in Matches, w in Weeks) m.isDivisional * Gain[w] * plays[m][w];
```

The solution

In order to format the solution for output, a set of tuples is created in which each team in the entire league has a unique identifier.

```
/// Map unique team id to team name for formatted solution.
tuple teamMapping{
  key int id;
  string name;
};
{teamMapping} teamLeague = {<t, item(teamDiv1,t)> | t in 1..nbTeamsInDivision} union
                           {<t+nbTeamsInDivision, item(teamDiv2,t)>
                             | t in 1..nbTeamsInDivision};
```

After the problem has been solved, the solution is printed using the `teamLeague` tupleset defined in the model file.

```
/// Postprocess to output a formatted solution.
tuple Solution {
  int    week;
  int    isDivisional;
  string team1;
  string team2;
};
sorted {Solution} solution = {<w, m.isDivisional,
                               item(teamLeague, <m.team1>).name,
                               item(teamLeague, <m.team2>).name>
                              | m in Matches, w in Weeks : plays[m][w] == 1};
```

The Result file contains the decision variable values in a table`plays` as well as this output table `solution`.

The following post-processing script in the model file produces the solution output in a week by week report.

```
execute DEBUG {
  var week = 0;
  writeln("Intradivisional games are marked with a *");
  for (var s in solution) {
    if (s.week != week) {
      week = s.week;
      writeln("================================");
      writeln("On week " + week);
    }
    if ( s.isDivisional ) {
      writeln(" *" + s.team1 + " will meet the " + s.team2);
    }
    else {
      writeln("  " + s.team1 + " will meet the " + s.team2)
    }
  }
}
```

The results of this can be seen in the Log file. For example for week 15:

```
On week 15
*Atlanta Falcons will meet the Dallas Cowboys
*Carolina Panthers will meet the New Orleans Saints
*Cincinnati Bengals will meet the Tennessee Titans
```

```
*Cleveland Browns will meet the Jacksonville Jaguars
*Detroit Lions will meet the Tampa Bay Buccaneers
*Green Bay Packers will meet the Minnesota Vikings
*Houston Texans will meet the Buffalo Bills
*Pittsburgh Steelers will meet the Indianapolis Colts
```

The optimal solution value for this model is 4448.

Possible modifications

To modify this example, try changing the number of teams in a division. Another possible modification is to change the number of games that must be played between teams of the same division.