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 CP; the file `Sport-CPLEX.pdf` presents the same problem using IP.

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-scheduling.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 * scheduleParams.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));
```

Modeling the variables and expressions

A solution to the problem can be modeled by assigning an opponent to each team for each week. Thus, the main decision variables in this model are indexed on the teams and weeks and take a value in `1..nbTeams`. The value at the solution of the decision variable `plays[t][w]` indicates the team that team `t` plays in week `w`.

```
/// Game variables - value of plays[t][w] will be the team assigned to play team t in week w.
dvar int plays[Teams][Weeks] in Teams;
```

Modeling the constraints

For each week and each team, there is a constraint that the team cannot play itself. Also, the variables must be constrained to be symmetric. If team `t` plays team `t2` in week w, then team t2 must play team t in week w. In constraint programming, it is possible to use a decision variable to index an array using an element expression.

```
forall (t in Teams, w in Weeks) {
  // A team cannot play itself.
  cannotPlaySelf:
  plays[t][w] != t;
  // The plays function is symmetrical.
  symmetricalPairs:
  plays[plays[t][w]][w] == t;
}
```

Each week, every team must be assigned to at most one game. To model this, the specialized alldifferent constraint is used. For a given week w, the values of play[t][w] must be unique for all teams t.

```
// Each week, each team is assigned to one game.
forall (w in Weeks)
  playsExactlyOnce:
  allDifferent( all (t in Teams) plays[t][w] );
```

One set of constraints is used to ensure the solution satisfies the number of intradivisional and interdivisional games that each team must play. The specialized expression count is used to "count" how many times each pair of teams plays a game against each other.

```
 // Each team plays the required number of (intra/inter) divisional matches.
 forall (t1 in Teams, t2 in Teams:  t1 < t2)
correctNumberOfGames:
   count( all(w in Weeks) plays[t1][w], t2 ) ==
        (intraDivisionalPair[t1][t2] == 1 ? nbIntraDivisional : nbInterDivisional);
```

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, t in Teams)
  cannotPlayAgain:
  if ( w < nbWeeks ) plays[t][w] != plays[t][w+1];
```

Each team must play at least a certain number of intradivisional games, nbFirstHalfGames, in the first half of the season. The specialized expression count is used to "count" how many times a pair of intradivisional teams play against each other in the first half of the season.

```
  // Some intradivisional games should be in the first half.
  forall (t1 in Teams)
   inDivisionFirstHalf:
   sum (t2 in Teams :  intraDivisionalPair[t1][t2] == 1)
  count (all(w in FirstHalfWeeks) plays[t1][w], t2 )
>= 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. An indicator matrix, intraDivisionalPair, is used to specify whether a pair of teams is in the same division or not. For each pair which is intradivisional, the incentive, or gain, is a power function of the week.

```
//// Gain is for intradivisional pairings only.
int intraDivisionalPair[ t1 in Teams][t2 in Teams ] =
  ( ((t2 <= nbTeamsInDivision) && (t1 <=nbTeamsInDivision)) ||
    ((t1 > nbTeamsInDivision) && (t2 > nbTeamsInDivision)) ) ? 1 : 0 ;

/// The goal is for intradivisional games to be played late in the schedule.
/// Only intradivisional pairings contribute to the overall gain.
int Gain[t1 in Teams][t2 in Teams][w in Weeks] =
  ((intraDivisionalPair[t1][t2]==1) ? w*w : 0) ;

/// The objective is used to maximize the overall quality of solutions.
dexpr int DivisionalLateness =
  sum(t in Teams, w in Weeks) Gain[t][plays[t][w]][w];
```

These cost functions are used to create an expression that models the overall cost. The cost here is halved as the incentive for each game gets counted twice.

```
maximize DivisionalLateness/2;
```

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,
                               intraDivisionalPair[t][plays[t][w]],
                               item(teamLeague, <t>).name,
                               item(teamLeague, <plays[t][w]>).name>  |
                              t in Teams, w in Weeks : t < plays[t][w]};
```

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
 Cincinnati Bengals will meet the Dallas Cowboys
 Cleveland Browns will meet the Detroit Lions
*Green Bay Packers will meet the Atlanta Falcons
*Houston Texans will meet the Indianapolis Colts
*Jacksonville Jaguars will meet the Buffalo Bills
*Minnesota Vikings will meet the Carolina Panthers
*New Orleans Saints will meet the Tampa Bay Buccaneers
*Pittsburgh Steelers will meet the Tennessee Titans
```

A time limit of 90 seconds has been set on the solve; thus, the best solution found may vary. 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.