Sports Scheduling in Video Games

Richard Gibson

April 24, 2009

Abstract

This paper investigates the problem of creating schedules on-line for sports video games, where the user specifies the number of teams and games played. We detail and implement a program using answer set programming which can find such schedules efficiently for a hypothetical NHL video game while maintaining the general format of real NHL schedules.

1 Introduction

Today, there are many sports video games which allow a human player, which we call the "user" throughout this paper, to take control of a professional team and play through an entire or abbreviated season of games against computer-controlled teams. However, the options available to the user are often limited and do not allow for fully personalized season schedules. For instance, there are baseball games where the user may only choose from a few set numbers of games between 14 and 162. In these games, there is no ability to choose, for instance, how many times each divisional opponent is faced or how many inter-league games are played, let alone the number of teams to use in the season. In certain hockey games, the situation is even worse; in season mode, the user may only play the full 30-team, 82-game (i.e. each team plays 82 games) hard-coded schedule that was set out by the real NHL before the video game was shipped. This results in long, tedious seasons that the user may never finish.

With today's technologies, sports video games should allow the user to tweak the schedule specifications to his liking. For instance, the user should be able to specify the number of teams to use in the season, as well as the number of games each team plays. In this paper, we focus on creating schedules for a hypothetical NHL hockey video game on-line in a short period of time. We implement a program using answer set programming which finds schedules that both adhere to the user's specifications, as well as resemble the format of real NHL schedules. The efficiency of our implementation is important; if starting a new season in the video game means hours of loading time, the user may often be too impatient to wait and thus never even play the season mode. We consider about five minutes to be about as long as the average video gamer would be willing to wait for a season to load.

While other sports scheduling problems have been studied, we believe that our particular work here is original. Firstly, there are studies focusing on scheduling for round-robin leagues, where all teams play each other an equal number of times and each team plays every round; the NHL's schedule is more general as each team does not play every other team an equal number of times. In the round-robin case, one can partition the problem by first finding a graph-theoretical arrangement of matches (which can be done in polynomial time), and then assign teams to the different placeholders so that constraints are satisfied and objectives are maximized (which is NP-hard) [Sch99]. Second, we are unaware of any work regarding scheduling for video games.

Typically in professional sports leagues like the NHL, as well as in sports video games, each game has a home team and an away team where the game is played at the home team's venue. Because of this, in the real world there are travel costs to minimize and fan attendance to maximize, as well as many other additional constraints. Thus the real-life scheduling problem is truly an optimization problem, which often takes several months of work just to find an acceptable sub-optimal solution. Some work in this area includes [SMYM06] where the cost problem is studied by separating the assignment of matches from the home and away assignments for those matches. For video games, we are simply looking for a feasible solution rather than an optimal solution. Since issues such as travel costs do not exist in the virtual world, we do not have to worry about optimizing an objective function, but rather just that the arrangement of matches be valid and "interesting." Finally, while integer programming [Tri02] and other constraint programming approaches [Sch99] have been considered, we are not aware of any uses of answer set programming for sports scheduling.

The remainder of this paper is organized as follows: Section 2 introduces the parameters of the schedule that the user may change and the constraints we apply to mimic real NHL scheduling formats. In Section 3, we present and explain our implementation and show that it is efficient even for schedules the size of the real-life NHL counterparts. Finally, in Section 4, we conclude our findings and propose some further work.

2 League Scheduling

A league is made up of n teams, where each team belongs to one of c conferences. Each conference of teams is further partitioned into v divisions. Teams in the same division play x games against each other, teams in the same conference but different divisions play y games against each other, and teams in different conferences play z games against each other (typically $x \geq y \geq z$, though we do not enforce this). For simplicity, we assume that each of the c conferences are of equal size, and that each of the v divisions are also of equal size; otherwise, we would need x, y, and z to be dependent on each division so that each team plays an equal number of games in total. In addition, for each game one team is denoted as the home team and the other as the away team. We require that each team plays an equal number of home games as away games. Again for simplicity, we will restrict x, y, and z to be even. This way, we can enforce that for two teams T and S, T is the home team for exactly half of the games against S (and vice versa). Finally, each game of the form S-at-T (thus denoting S as the away team and T as the home team) is assigned to one of d days so that no team plays more than one game per day. An assignment of all the games in this manner is called a *minimum schedule*.

Intuitively, the problem of finding a minimum schedule is difficult. We can think of this as a restricted matching problem of a bipartite graph; here, the problem is to find a complete matching (i.e. a matching which covers one of the two vertex partitions) M of a bipartite graph $B = (V, E)$ such that given a collection of subsets $E_1, ..., E_k \subseteq E$ and positive integers $r_1, ..., r_k$, M satisfies $|M \cap E_j| \leq r_j$ for all $j = 1, ..., k$. For us, we can denote B as the complete bipartite graph where one partition of V is the set of games of the form $Game_j = T_{j1}$-at-T_{j2} to be played, and the other partition is $\{1, ..., d\} \times \{1, ..., n\}$ (n copies of each day). Then for each day $\ell \in \{1, ..., d\}$ and each pair of games $Game_i$ and $Game_j$, $i \neq j$, that have at least one team in common (i.e. $|\{T_{i1}, T_{i2}\} \cap \{T_{j1}, T_{j2}\}| \geq 1$), we create the restriction $E_{ij\ell} = \{Game_i(\ell, m), Game_j(\ell, p) \mid m, p \in \{1, ..., n\}\}$ with $r_{ij\ell} = 1$. Assuming that d is large enough for a minimum schedule to exist, this ensures that a solution M to this restricted matching problem

directly corresponds to a minimum schedule by assigning $Game_i$ to day ℓ where $Game_i(\ell, m) \in M$ for some $m \in \{1, ..., n\}$. As the restricted matching problem is NP-complete ([IRT78]), we can think of finding a minimum schedule, in a sense, as hard as well. Unfortunately, this is not enough to assert NP-completeness and we have yet to find a proof.

In the real NHL, $n = 30$ teams are separated into two fifteen-team conferences, where each conference contains three divisions of five teams each. In the regular season, each team plays $x = 6$ games against divisional opponents, $y = 4$ games against other teams in the same conference, and $z = 1$ game against inter-conference opponents, for a total of 82 games (each team plays an additional game against three inter-conference rival teams). Not surprisingly, each team plays 41 home games and 41 away games. These games are scheduled between early October and early-to-mid April, and so if we account for holidays and breaks such as Christmas and the NHL All-Star weekend where no games are scheduled, the number of days d is roughly twice the number of games each individual team plays. We will not consider creating schedules with more than 30 teams or 82 games per team as our goal is really to allow video games to have smaller schedules than real NHL schedules.

The NHL considers other issues besides optimizing travel distances and fan attendance when creating a regular season schedule. For instance, teams are not scheduled to play too many games in consecutive days so that players can rest, and a team's home games and away games are typically interleaved to some degree. Sports video game schedules should resemble their corresponding professional league schedules to give the user the feeling of realism. Therefore we consider similar constraints when creating an NHL video game schedule:

- No team plays more than G games per M consecutive days, where $(G, M) \in \{(2, 3), (3, 5), (5, 8)\}$. This ensures that a team's games are not too bunched together as player fatigue may have some effect in the video game.

- No team plays less than 1 game per 7 consecutive days. This constraint helps to spread out a team's games across all of the days in the schedule.

- At least one game is played each day. This further helps to spread out the games across all of the schedule's days.

- No two teams play each other more than H times per N consecutive days, where $(H, N) \in \{(2, 7), (3, 14)\}$. It is often less fun to play the same team repeatedly, and the "fun-factor" is certainly important for us.

- No team plays more than 5 consecutive home games or 5 consecutive away games. Similar to the previous constraint, it is more fun to alternate between home games and away games to some degree, rather than playing, for instance, all the home games before the away games.

- If $x > 2$, then teams play at least one *home-and-home series* against each team in their division. A home-and-home series is a pair of games between two teams over 2 consecutive days where each team is at home for one of the two games. Real NHL schedules often include home-and-home series between rival teams, so we do the same for our video game schedules.

Any minimum schedule that also satisfies the listed constraints above is called a *proper schedule*.

Our goal in this paper is to find a proper schedule for a league in a short length of time, where the size and length of the league is determined by the user. In particular, the user specifies the values of n, c, v, x, y, and z. The user also determines which n teams to include in the league as well as their arrangement into c conferences and v divisions. We make one more simplification by assuming that $n \geq 4$ and that each team plays at least 6 games. This is done since proper schedules of unusually small size may not exist; for example, there is no proper schedule for the values $n = 2$, $c = v = 1$, $x = 4$, and $d = 8$ because the two teams cannot play each other more than 3 times per 14 consecutive days.

3 Implementation and Experimental Results

We use answer set programming to find NHL video game schedules. All of our implementations use lparse version 1.0.5 ([Syr]) as a front-end for the answer set solvers used. In Section 3.1, we compare Smodels version 2.26 ([Smo]) and Clasp version 1.1.3 ([Cla]) at finding minimum schedules of various sizes using two different encodings, before tackling proper schedules in Section 3.2. All of our tests were run on a 2.00 GHz processor with 4 GB of memory. We note here that our implementations were tested with the newer versions of lparse (1.1.1) and Smodels (2.33) on another machine and were still found to give correct solutions.

3.1 Minimum Schedules: Smodels versus Clasp

Here, we describe our lparse code `MinScheduleBuiler.lp` which we use to find minimum schedules. First, teams are denoted by predicates of the form `team(teamName, conferenceName, divisionName)` and we list the teams in a separate file `Teams.lp`. Our remaining "base" predicates define the allowed days to schedule over, which team names are valid, and the home-away team designations:

```
day(1..d).

isTeam(T) :- team(T, Conf, Div).

location(home).

location(away).
```

Each possible game in our schedule is represented by a predicate `game(T1, T2, D, L)`, which is true when team `T1` plays team `T2` on day `D`, with `T1` at home if and only if `L = home`. We ensure that teams in the same division play each other exactly x times, with each team being at home for half of the games, with the following definition:

```
x/2{game(T1, T2, D, L) : day(D)}x/2 :-
    team(T1, Conf, Div),
    team(T2, Conf, Div),
    location(L),
    T1 < T2.
```

The definitions for intra-conference, inter-division games and for inter-conference games using y and z respectively are similar. Since a game between two teams is "symmetric" (`game(T1, T2, D, away)` is equivalent to `game(T2, T1, D, home)`), we find it useful to include the definition below:

```
game(T2, T1, D, L1) :-
    game(T1, T2, D, L2),
    isTeam(T1),
    isTeam(T2),
    location(L1),
    location(L2),
    L1 != L2,
    day(D).
```

Before our program can find a minimum schedule, we need to enforce that no team has more than one game scheduled per day. We experiment with two similar definitions for doing this. The first, which we denote as the "LHS definition," defines a cardinality constraint that ensures that each team has either 0 or 1 game each day:

```
0{game(T1, T2, D, L) : isTeam(T2) :  T1 != T2 :  location(L)}1 :-
    isTeam(T1),
    day(D).
```

The second, denoted the "RHS definition," simply moves the cardinality constraint in the LHS definition to the right-hand side and adjusts the cardinality appropriately:

```
:- 2{game(T1, T2, D, L) : isTeam(T2) :  T1 != T2 :  location(L)},
    isTeam(T1),
    day(D).
```

Including either of these two definitions in the code above is sufficient for finding a minimum schedule.

Note that the user specifies the teams to use in the schedule, as well as the values of x, y, and z. However, the value of d, the number of days to schedule over, is up to us to determine. So what should d be? As mentioned in Section 2, the real NHL uses a value of d that is about twice the number of games each individual team plays. Since our video game schedules should look similar in format to the real NHL schedules, we set d to exactly twice the number of such games in all of our experiments.

We compare the efficiencies of Smodels and Clasp at finding minimum schedules for five different examples. Example 1 is a small, 4-team, 6-game schedule which is the smallest sized schedule we wish our program to find (see Section 3). Example 2 is a medium-sized, 16-team, 36-game schedule and Example 3 is a large, 30-team, 82-game schedule (the largest we wish our program to solve). Examples 4 and 5 test cases where we have few teams but many games and many teams but few games respectively. Example 4 is a 6-team, 62-game schedule and Example 5 is a 24-team, 20-game schedule. The parameters used for each example are listed in Table 3.1. To run the program for, say, Example 1 with Smodels, we enter

```
lparse -c x=2 -c y=0 -c z=0 -c d=12 MinScheduleBuilder.lp Teams.lp | smodels
```

at the command line.

	n	c	v	x	y	z	d
Example 1	4	1	1	2	0	0	12
Example 2	16	2	2	4	2	2	72
Example 3	30	2	3	8	2	2	164
Example 4	6	1	2	16	10	0	124
Example 5	24	3	2	4	2	0	40

Table 3.1: The parameters used for each example of our experiments.

In every example, Clasp outperforms Smodels at finding a minimum schedule. This is not too surprising since Clasp has special functionality for dealing with cardinality constraints, which are included in four definitions of our encoding. However, it is surprising at how much more efficient Clasp is compared to Smodels for larger cases. In Example 3, it takes Smodels about 150 minutes to find a minimum schedule using the RHS definition, whereas Clasp using the RHS definition finds one in under two seconds. We are not aware of other problems where Clasp outperforms Smodels by this magnitude.

In general, the RHS definition appears to be more efficient than the LHS definition. Both Clasp and, in general, Smodels see speed-ups in computation time when using the RHS definition, and lparse is generally a bit faster at grounding the program with the RHS definition. We suspect that this is partially due to the sizes of the grounded programs as the RHS grounding is smaller (about 31MB compared to the LHS grounded program which is about 37MB for Example 3). Graphs of these results are displayed in Figure 3.1 below. For all future experiments, we use the RHS definition and Clasp as this combination performs the best.

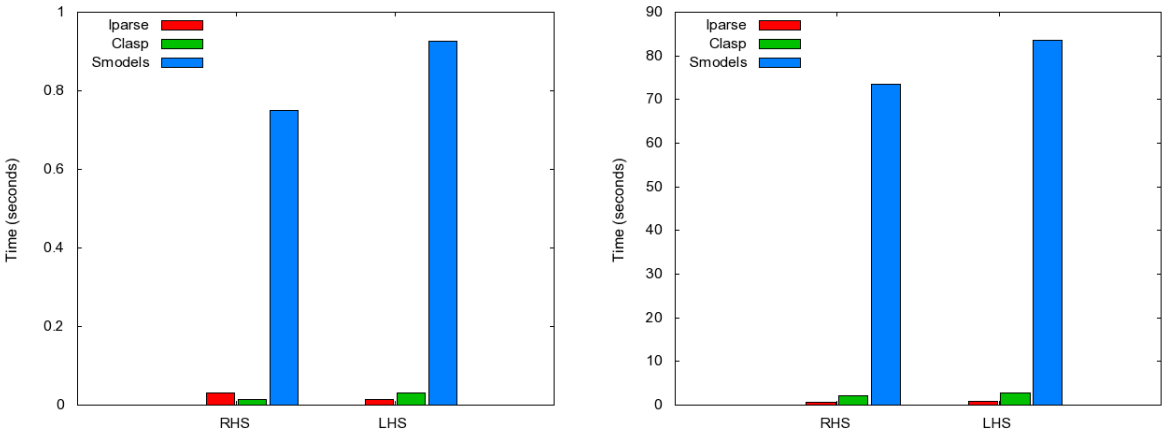
3.2 Proper Schedules

Being able to find minimum schedules in just seconds is a good step towards our true goal, as mentioned earlier, of finding proper schedules in a short amount of time. Since a proper schedule is also a minimum schedule, we reuse all of the code in `MinScheduleBuilder.lp` (with the RHS definition) from Section 3.1. We then extend our encoding to address each of the six additional constraints presented earlier in Section 2, which we detail below:

No team plays more than G games per M consecutive days, where $(\mathbf{G}, \mathbf{M}) \in \{(\mathbf{2}, \mathbf{3}), (\mathbf{3}, \mathbf{5}), (\mathbf{5}, \mathbf{8})\}$. To simplify this and future constraint encodings, we introduce a predicate `hasGame(T1, D)` which is true when team `T1` plays a game on day `D`:

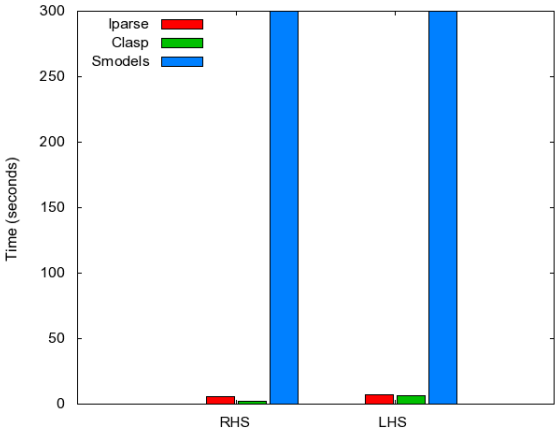
```
hasGame(T1, D) :-

    game(T1, T2, D, L),
    isTeam(T1),
    isTeam(T2),
    T1 != T2,
    day(D),
```

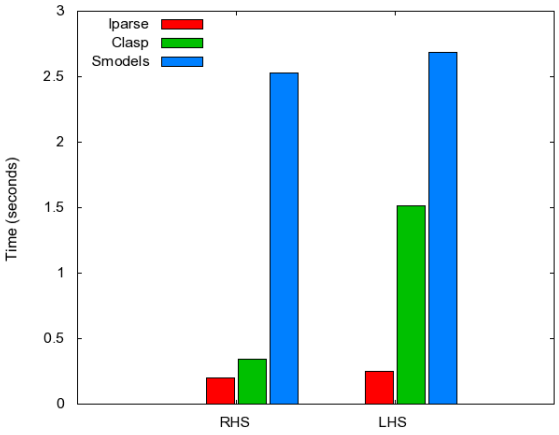


(a) Example 1

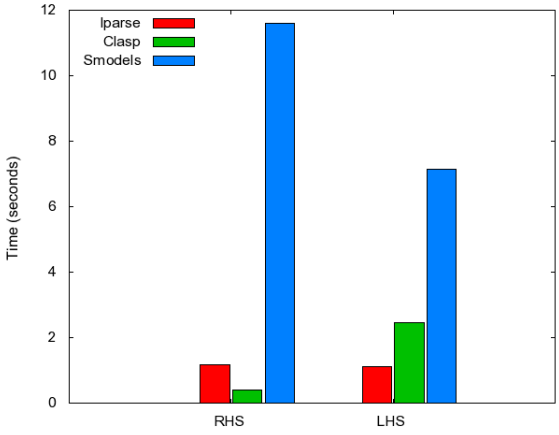
(b) Example 2



(c) Example 3



(d) Example 4



(e) Example 5

Figure 3.1: The computation times required by lparse to ground our program, and by Clasp and Smodels to find a minimum schedule in each of our five examples. Note that in Example 3, Smodels does not find a minimum schedule in less than five minutes.

```
    location(L).
```

Then given any window of consecutive days of length M, we can simply use a cardinality constraint to ensure that it is impossible for a team to play more than G games in the window:

```
:- G+1{hasGame(T1, D) : day(D) : Dmin <= D : D <= Dmax},
    isTeam(T1),
    day(Dmin),
    day(Dmax),
    Dmin < Dmax,
    Dmax - Dmin == M-1,
    atMost(G, M).

atMost(2, 3).

atMost(3, 5).

atMost(5, 8).
```

If we ever want to change the domain of (G, M) in our definition of this constraint, our encoding can easily be modified by adding or removing the `atMost` predicates. Note that we could have introduced a LHS version of this instead, similar to what was done in Section 3.1; however, RHS versions were found to be faster for this constraint and the below constraints as well. We omit the details of these experiments here.

No team plays less than 1 game per 7 consecutive days. We can encode this constraint simply by adjusting the cardinalities and window sizes of the previous contraint:

```
:- {hasGame(T1, D) : day(D) : Dmin <= D : D <= Dmax}G-1,
    isTeam(T1),
    day(Dmin),
    day(Dmax),
    Dmin < Dmax,
    Dmax - Dmin == M-1,
    atLeast(G, M).

atLeast(1, 7).
```

No two teams play each other more than H times per N consecutive days, where $(\mathbf{H}, \mathbf{N}) \in \{(\mathbf{2}, \mathbf{7}), (\mathbf{3}, \mathbf{14})\}$. This constraint is encoded similar to the previous two, except that we cannot use the convenient `hasGame` predicate here:

```
:- G+1{game(T1, T2, D, L) : location(L) : day(D) : Dmin <= D : D <= Dmax},
    isTeam(T1),
```

```
    isTeam(T2),
    T1 != T2,
    day(Dmin),
    day(Dmax),
    Dmin < Dmax,
    Dmax - Dmin == M-1,
    diverse(G, M).

diverse(2, 7).

diverse(3, 14).
```

At least one game is played each day. Again, we keep the left-hand side empty:

```
:- {hasGame(T1, D) : isTeam(T1)}0,
    day(D).
```

If x > 2, then teams play at least one home-and-home series against each team in their division. We find it convenient to split this constraint into two parts. The first defines a predicate `homeAndHome(T1, T2)` which is true when $x > 2$, teams T1 and T2 are in the same division, and they play a home-and-home series against each other:

```
homeAndHome(T1, T2) :-
    team(T1, Conf, Div),
    team(T2, Conf, Div),
    T1 != T2,
    game(T1, T2, D1, L1),
    game(T1, T2, D2, L2),
    day(D1),
    day(D2),
    abs(D2 - D1) == 1,
    location(L1),
    location(L2),
    L1 != L2,
    x > 2.
```

We then reject any schedules missing home-and-home series between appropriate teams:

```
:- team(T1, Conf, Div),
    team(T2, Conf, Div),
    T1 != T2,
```

```
not homeAndHome(T1, T2),
x > 2.
```

Our encoding to this point, which we call `ProperScheduleBuilderPart1.lp (PSBP1.lp)` is fairly efficient at finding schedules subject to the constraints included. The first two columns of Table 3.2 below show the computation times for both lparse and Clasp for each of our five example instances from Table 3.1. The most difficult instance for `PSBP1.lp` was Example 4, but lparse and Clasp can still find a schedule in under two and a half minutes. However, these schedules may not be proper schedules as we have yet to address the constraint that "no team plays more than 5 consecutive home games or 5 consecutive away games." Unfortunately, we could not find any way of extending `PSBP1.lp` to incorporate this constraint without dramatically increasing computation time for larger instances, particularly for Example 3.

To ensure that we eventually find a proper schedule, we create a second lparse encoding named `ProperScheduleBuilderPart2.lp (PSBP2.lp)`. This encoding takes as input the schedule found by `PSBP1.lp` and reconfigures the home and away team assignments for each game until we have a proper schedule. The mechanics of this are a little tricky, and the full encoding of `PSBP2.lp` with comments is produced in Appendix A.

We now have a program capable of finding proper schedules. Note that `PSBP2.lp` requires the output of `PSBP1.lp` in a format that lparse can understand. Thus, we write a small, simple parser in Java called `ScheduleParse.java` which simply takes the Clasp output from `PSBP1.lp`, assumed to be in a file called `games.lp`, and writes the `game(T1, T2, D, L)` predicates found in the stable model to a file called `parsedGames.lp`. Thus to run our program, for instance on Example 3, we require three commands (where again `Teams.lp` contains the correct number and arrangement of teams):

```
lparse -c d=164 -c x=8 -c y=2 -c z=2 ProperScheduleBuilderPart1.lp Teams.lp
| clasp-1.1.3 > games.lp

java ScheduleParse

lparse -c d=164 -c x=8 -c y=2 -c z=2 ProperScheduleBuilderPart2.lp Teams.lp
parsedGames.lp | clasp-1.1.3
```

Then each predicate of the form `matchup(T1, T2, D)` (see Appendix A) in the stable model of `PSBP2.lp` represents the game `T1`-at-`T2` on day `D`. The code for `ScheduleParse.java` is straightforward and is omitted here.

We run our program on each of our example instances in Table 3.1 and the results are shown in Table 3.2 below. To our delight, a proper schedule can be found in any example in under three and a half minutes. As expected, Example 3 requires the most computation time as it is the largest instance. However, note that Example 4 requires more time than Example 3 (lparse plus Clasp) to find a solution for `PSBP1.lp`. This is likely because the constraint "no two teams play each other more than H times per N consecutive days" is more difficult to satisfy when the number of teams is small and the number of games is large. Some additional experiments have shown that the obscure 4-team, 82-game case has unacceptably long computation times for `PSBP1.lp` (over five minutes). For now, we recommend that a video game not allow the option of a league with such a large number of games when the number of teams is very small. We address this further in Section 4.

	lparse - `PSBP1.lp`	Clasp - `PSBP1.lp`	lparse - `PSBP2.lp`	Clasp - `PSBP2.lp`	Total
Example 1	0.062	0.031	0.093	0	0.326
Example 2	5.413	1.904	2.792	0.375	18.455
Example 3	62.259	43.057	44.007	60.451	209.774
Example 4	1.747	135.814	0.936	0.187	138.684
Example 5	5.366	60.825	2.964	0.250	69.405

Table 3.2: The computation times, in seconds, required for lparse to ground our encodings, and for Clasp to find a stable model of the grounded instances. The stable model found by `ProperScheduleBuilderPart2.lp` (`PSBP2.lp`) is a proper schedule, and the total time required to find a proper schedule for each example is given.

4 Conclusion

We have successfully shown that NHL video game schedules with various numbers of teams and games can efficiently be found on-line. In addition, answer set programming using lparse and Clasp were sufficient for this task, and we argue that future NHL video games should incorporate schedule customization. While our encodings presented require reasonably short computation times, we did attempt to increase efficiency even further. For instance, we tried simpler encodings for `PSBP1.lp` that did not incorporate home or away team distinctions, since `PSBP2.lp` ignore these anyways. Unfortunately, the results of these efforts were unexpectedly less efficient and thus were abandoned. However, we do not doubt that there is room for improvement in the efficiency of our program.

While we only focused on finding schedules for NHL video games, we can also address scheduling for video games of other professional sports leagues. The NBA basketball schedule is very similar to the NHL schedule; the 30-team, 82-game NBA regular season schedule takes place between early November and mid-to-late April where like the NHL, teams are split among two conferences of three divisions each. This suggests that our program would also be appropriate to use in an NBA video game for on-line, customizable scheduling. However, the MLB baseball regular season schedule is a 30-team, 162-game schedule from early April to late September. This means that the MLB has a higher games-per-team to days ratio than the NHL and NBA (days off for a team are much more rare in the MLB), which would make the search for a schedule harder if we adjust the value of d in our program accordingly. Also, MLB games are usually grouped into three or four game series between the same two teams at the same venue, and any scheduling program for an MLB video game should mimic this type of format. Creating such a program is certainly of interest for future work.

In addition to MLB scheduling, there are a number of issues worth further investigation. Firstly, we expect that our program, with a few appropriate changes and additions, should be capable of removing the simplifications we imposed in Section 2. In particular, we should be able to address the cases where conferences and divisions are unevenly sized and the values of x, y, and z are not necessarily even. This would mean defining x, y, and z values specific to each division and enforcing teams to play an equal number of total away games and home games (plus or minus 1) rather than just for games between each pair of teams. Next, we could include some special cases to remove or adjust certain constraints when searching for schedules in extreme

cases, such as the 4-team, 82-game case. These special rules would look similar to the home-and-home series condition where we assert that $x > 2$, and may allow these extreme cases to be solved in a reasonable amount of time. Finally, we note here that it seems extremely unlikely, but perhaps possible, that the stable model found by `PSBP1.lp` would not lend itself to a proper schedule after reassigning home and away team distinctions. In this scenario, `PSBP2.lp` would be unable to find a solution, so what should we do? We could simply resort to the schedule corresponding to the output of `PSBP1.lp` as it is "nearly" a proper schedule, or request a new stable model from `PSBP1.lp`, or even increase the number of allowed consecutive home or away games for one team. However, this scenario appears extremely unprobable as it was never encountered in any of our experiments, and since we already have some strategies for dealing with this unlikely event, it should be considered low priority.

Appendix

A Encoding of `ProperScheduleBuilderPart2.lp`

Below is the encoding used for `ProperScheduleBuilderPart2.lp` (see Section 3.2):

```
% Remove the old location assignment

game(T1, T2, D) :-

    game(T1, T2, D, L),
    isTeam(T1),
    isTeam(T2),
    day(D),
    location(L).

% The assignedGame predicate has the same meaning as the game predicate from

% the first part of this program.

% assignedGame(T1, T2, D, L) is true only when game(T1, T2, D) is true and

% when T1 is assigned to be the home (away) team where L = home (away).

x/2{assignedGame(T1, T2, D, L) : game(T1, T2, D)}x/2 :-

    team(T1, Conf, Div),
    team(T2, Conf, Div),
    location(L),
    T1 < T2.

y/2{assignedGame(T1, T2, D, L) : game(T1, T2, D)}y/2 :-

    team(T1, Conf, Div1),
    team(T2, Conf, Div2),
    location(L),
```

```
    T1 < T2,
    Div1 != Div2.

z/2{assignedGame(T1, T2, D, L) : game(T1, T2, D)}z/2 :-
    team(T1, Conf1, Div1),
    team(T2, Conf2, Div2),
    location(L),
    T1 < T2,
    Conf1 != Conf2.

% Each game must be assigned to exactly one location

1{assignedGame(T1, T2, D, L) : location(L)}1 :-
    game(T1, T2, D),
    isTeam(T1),
    isTeam(T2),
    day(D),
    T1 < T2.

% Assignments are symmetric

assignedGame(T2, T1, D, L2) :-
    assignedGame(T1, T2, D, L1),
    isTeam(T1),
    isTeam(T2),
    day(D),
    location(L1),
    location(L2),
    L1 != L2.

% hasGame(T1, D) is true when T1 plays a game on day D.

hasGame(T1, D) :-
    game(T1, T2, D),
    isTeam(T1),
    isTeam(T2),
    day(D).

% hasGameAt(T1, D, L) is true when team T1 plays a game at L on day D.

hasGameAt(T1, D, L) :-
```

```
    assignedGame(T1, T2, D, L),
    isTeam(T1),
    isTeam(T2),
    T1 != T2,
    day(D),
    location(L).
% hasPlayedNInARowAtOneLoc(T1, N, D, L) is true when at the end of day D, the
% previous N games played by T1 were all at location L.
% Initially 0 games have been played (putting home here is arbitrary).
hasPlayedNInARowAtOneLoc(T1, 0, 0, home) :-
    isTeam(T1).
% Still no game has been played
hasPlayedNInARowAtOneLoc(T1, 0, D, home) :-
    isTeam(T1),
    day(D),
    not hasGame(T1, D),
    hasPlayedNInARowAtOneLoc(T1, 0, D-1, home).
% If no game was played on day D, then simply use the truth value at day D-1.
hasPlayedNInARowAtOneLoc(T1, N, D, L) :-
    isTeam(T1),
    consecLocCounter(N),
    day(D),
    location(L),
    not hasGame(T1, D),
    hasPlayedNInARowAtOneLoc(T1, N, D-1, L).
% If a game was played on day D at the same location as the previous game,
% then we must increment the number of consecutive games played at the appropriate
% location.
hasPlayedNInARowAtOneLoc(T1, N, D, L) :-
    isTeam(T1),
    consecLocCounter(N),
    day(D),
```

```
    location(L),
    hasGameAt(T1, D, L),
    hasPlayedNInARowAtOneLoc(T1, N-1, D-1, L).
% If a game was played on day D at a different location to the previous game,
% then we have only played 1 consecutive game at the new location.
hasPlayedNInARowAtOneLoc(T1, 1, D, L1) :-
    isTeam(T1),
    day(D),
    location(L1),
    hasGameAt(T1, D, L1),
    hasPlayedNInARowAtOneLoc(T1, N, D-1, L2),
    consecLocCounter(N),
    location(L2),
    L1 != L2.
% No team can play more than N consecutive games at one location, where consecLoc(N).
:- isTeam(T1),
    consecLoc(N),
    location(L),
    hasPlayedNInARowAtOneLoc(T1, N+1, D, L),
    day(D).
% homeAndHome(T1, T2) is true if T1 and T2 are in the same division, x > 2,
% and T1 and T2 play a home-and-home series against each other.
homeAndHome(T1, T2) :-
    team(T1, Conf, Div),
    team(T2, Conf, Div),
    T1 != T2,
    assignedGame(T1, T2, D1, L1),
    assignedGame(T1, T2, D2, L2),
    day(D1),
    day(D2),
    abs(D2 - D1) == 1,
    location(L1),
    location(L2),
```

```
    L1 != L2,
    x > 2.
% If two teams are in the same division and play each other more than twice,
% they play at least one home and home series.
:- team(T1, Conf, Div),
    team(T2, Conf, Div),
    T1 != T2,
    not homeAndHome(T1, T2),
    x > 2.
% matchup(T1, T2, D) says T1 plays at T2 on day D (this is just to simplify
% output).
matchup(T1, T2, D) :-
    assignedGame(T1, T2, D, away),
    isTeam(T1),
    isTeam(T2),
    day(D).
matchup(T2, T1, D) :-
    assignedGame(T1, T2, D, home),
    isTeam(T1),
    isTeam(T2),
    day(D).
% Facts
day(1..d).
isTeam(T) :- team(T, Conf, Div).
location(home).
location(away).
consecLoc(5).
consecLocCounter(0..6).
```

References

[Cla] (http://www.cs.uni-potsdam.de/clasp).

[IRT78] A. Itai, M. Rodeh, and S.L. Tanimoto, *Some matching problems for bipartite graphs*, Journal of the Association for Computing Machinery **25** (1978), 517–525.

[Sch99] A. Schaerf, *Scheduling sport tournaments using constraint logic programming*, Constraints: An International Journal **4** (1999), 43–65.

[Smo] (http://www.tcs.hut.fi/Software/smodels).

[SMYM06] A. Suzuka, R. Miyashiro, A. Yoshise, and T. Matsui, *Dependent randomized rounding to the home-away assignment problem in sports scheduling*, IEICE Trans. Fundamentals **E89-A** (2006), 1407–1416.

[Syr] T. Syrjänen, *Lparse 1.0 user's manual*, (http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz).

[Tri02] M.A. Trick, *Integer and constraint programming approaches for round-robin tournament scheduling*, PATAT (2002), 63–77.