Integer and Constraint Programming Approaches for Round Robin Tournament Scheduling

Michael A. Trick

Graduate School of Industrial Administration, Carnegie Mellon, Pittsburgh, PA USA, 15213 trick@cmu.edu

Abstract. Real sports scheduling problems are difficult to solve due to the variety of different constraints that might be imposed. Over the last decade, through the work of a number of researchers, it has become easier to solve round robin tournament problems. These tournaments can then become building blocks for more complicated schedules. For example, we have worked extensively with Major League Baseball on creating "what-if" schedules for various league formats. Success in providing those schedules has depended on breaking the schedule into easily solvable pieces. Integer programming and constraint programming methods each have their places in this approach, depending on the constraints and objective function.

1 Introduction

There has been a lot of recent work on combinatorial optimization methods for creating sports schedules. Much of this work has revolved around creating single round-robin schedules, where every team plays every other team once, and double round-robin schedules, where every team plays every other team twice (generally once at its home venue and once at each opposing venue).

Round-robin scheduling is interesting in its own right. Some leagues have a schedule that is a single or double round-robin schedule. Examples of this include many U.S. college basketball leagues and many European football (soccer) leagues. For such leagues, the scheduling problem is exactly a constrained round-robin scheduling problem, where the constraints are generated by team requirements, league rules, media needs, and so on.

For other leagues, the schedule is not a round-robin schedule but it can be divided into sections that are round-robins among subsets of teams. There are good reasons, both algorithmically and operationally, to make these divisions. By creating these sections, schedulers are able to use results about round-robin schedules to optimize the pieces. Operationally, such schedules are often appealing to the leagues since they offer an understandable structure and often are perceived as being fairer than unstructured schedules. For instance, by scheduling four consecutive round-robins, each team will see each other in each quarter of the schedule; an unstructured schedule would not necessarily require that.

This work was originally motivated by by looking at a large practical scheduling problem, that of Major League Baseball (MLB). Fully defining the MLB schedule is a daunting task, requiring the collation of more than 100 pages of team requirements and requests, along with an extensive set of league practices. The key insight into effectively scheduling MLB, however, was the recognition that the complicated schedule could generally be broken into various phases, where each phase consists of a round-robin schedule, sometimes among subsets of teams. These round-robin schedules often have additional constraints reflecting team requirements or effects from other phases of the schedule. By better understanding constrained round-robin scheduling, we are better able to schedule MLB.

2 Round Robin Scheduling

We begin with the most fundamental problem in sports scheduling: designing an unconstrained round robin schedule. In a round robin schedule, there are an even number n teams each of whom plays each other team once over the course of the competition. We will work only with *compact* schedules: the number of slots for games equals $n-1$, so every team plays one game in every slot. We refer to this as a Single Round Robin (SRR) Tournament.

A related problem is the Bipartite Single Round Robin (BSRR) Tournament problem. Here, the teams are divided into two groups X and Y, each with $n/2$ teams. There are $n/2$ slots during which all teams in X need to play all teams in Y, but teams within X (and within Y) do not play each other.

Henz, Müller, and Thiel [12] (who we will refer to as HMT) have recently examined SRR carefully in the constraint programming context. We expand on their work by examining integer programming formulations and further exploring the strengths and weaknesses of the different approaches.

HMT point out that the SRR can be formulated with two major types of constraints. First, the games in every slot correspond to a one-factor (or matching) of the teams. Second, for any team i, its opponents across all of the slots must be exactly the set of teams except for i. We will call the first the *one-factor constraint* and the second the *all-different constraint.* Different formulations have different ways of encoding and enforcing these constraints.

3 Integer Programming Formulation

Our basic integer program for SRR begins with binary variables x_{ijt} which is 1 if teams i and j play each other in slot t, and is 0 otherwise. Since the order of i and j does not matter, we could either define this only for $i > j$ or set $x_{ijt} = x_{jit}$ for all i, j, t.

This leads to the formulation (in the OPL language[25]):

```
int n=...;
range Teams [0..n-1];
range Slots [1..n-1];
range Binary 0..1;
var Binary plays[Teams,Teams,Slots];

solve {
   //No team plays itself
   forall (i in Teams, t in Slots) plays[i,i,t] = 0;


   //Every team plays one game per slot
   //One-factor constraints
   forall (ordered i,j in Teams, t in Slots)
      plays[i,j,t] = plays[j,i,t];
   forall (i in Teams, t in Slots)
      sum (j in Teams) plays[i,j,t] = 1;

   //Every team plays every other team
   //All-different constraint
   forall (i,j in Teams: i<>j) sum(t in Slots) plays[i,j,t] = 1;

};
```

We call the above the Base-IP Formulation. We can strengthen this formulation by a stronger modeling of the one-factor constraint. The polyhedral structure of the one-factor polytope is perhaps the most well-studied polytope in combinatorial optimization. Edmonds [9] showed that the polytope is defined by adding *odd-set* constraints. In this context, the odd set constraints are as follows. For a particular slot t, let S be a set of teams, $|S|$ odd. Then,

$$sum_{i \in S, j \notin S} x_{ijt} \geq 1$$

is valid for the one-factor constraint. If we add all of these constraints, then the one-factor constraint is precisely defined in the polyhedral sense (all extreme points of the polyhedron are integer).

We call the formulation with all of the odd-set constraints the Strong-IP formulation. Of course, there are too many odd-set constraints to simply add them all to the integer program. We can solve Strong-IP by using a constraint generation method. In this method, we begin with a limited set of odd-set constraints and solve the linear relaxation of the instance. We then identify violated odd-set constraints (this can be done with a method by Padberg and Rao [16] using cut-trees) and add them to the formulation. We repeat until either we have added "enough" constraints or until all odd-set constraints are satisfied. At that point, we then continue our normal branch and bound approach to integer programs.

Strong-IP is not needed for the BSRR problem. In this case, the odd set constraints are redundant, so need not be added.

4 Constraint Programming Formulation

HMT extensively analyze constraint programming formulations for SRR. Their basic variables are `opponent[i,t]` which gives the opponent i plays in slot t. Their formulation can be represented very simply:

```
int n=...;
range Teams [0..n-1];
range Slots [1..n-1];
var Teams opponent[Teams,Slots];

solve {
   //No team plays itself
   forall (i in Teams, t in Slots) opponent[i,t] <>i;

   //Every team plays one game per slot
   //One-factor constraint
   forall (t in Slots)
      one-factor(all (i in Teams) opponent[i,t]);

   //Every team plays every other team
   //All-different constraint
   forall (i in Teams)
      all-different(all (t in Slots) opponent[i,t]);

};
```

The key issue is to define how the one-factor and all-different constraints are implemented. There are a variety of propagation algorithms available for each. HMT argues convincingly that the all-different constraint should use arc-consistent propagation by the method of Régin [17]. Briefly, arc-consistency means that the domains of the variables are such that for any value in a domain, setting the variable to that value allows settings for all the other variables so that the constraint is satisfied. For more details on the fundamentals of constraint programming, see [14].

For the one-factor constraint, the main emphasis of HMT, they examine three different approaches. The first is the simplest. It uses the constraints:

```
  forall (i in Teams)
     opponent[opponent[i,t]] = i;
```

This set of constraints is sufficient to define the one-factor constraint, but its propagation properties are not particularly strong. In particular, the only domain reduction that is done is when i is in the domain for j for a particular time period t, but j is not in the domain of i for that time period. In that case, i can be removed for the domain of j.

The propagation properties of this constraint can be improved by adding the redundant constraint:

```
all-different(all (i in Teams) opponent[i,t]);
```

HMT give an example where adding the all-different constraint leads to improved domain reduction. We call this combination the *all-different* one-factor approach.

The combination of these constraints do not create an arc-consistent propagation for the one-factor constraint. HMT provide an arc-consistent propagation method using results from non-bipartite matchings. In general, this approach is much better than the all-different approach. The exceptions mimic when the Strong-IP does not improve on Basic-IP: if the underlying graph is bipartite, then the all-different approach is arc-consistent.

To prove this, let D_i be the feasible opponents for i. We say the D_i are bipartite if we can divide the teams into X and Y such that

1. $|X| = |Y| = n/2$
2. $i \in X \rightarrow D_i \subseteq Y$
3. $i \in Y \rightarrow D_i \subseteq X$

Theorem 1. *If the D_i are bipartite, then arc-consistency for the constraints*

```
forall (i in Teams)
  opponent[opponent[i]] = i;
all-different(opponent[i]);
```

implies arc-consistency for `one-factor(opponent)`

Proof. Suppose the D_i are consistent for the constraints

```
forall (i in Teams)
  opponent[opponent[i]] = i;
all-different(opponent);
```

Let $j \in D_i$. We will show there is a one-factor that has j as the opponent for i. Without loss of generality, we will assume $j \in Y$, so $i \in X$. By consistency of the all-different constraint, there is a setting of opponent values so that `j=opponent[i]`, and `all-different[opponent]`. Create a new setting of the opponent values `opponent'` such that `opponent'[i] = opponent[i]` for $i \in X$ and `opponent'[i] = opponent[opponent[i]]` for $i \in Y$. By the consistency requirement, `opponent'[i]` $\in D_i$ for all i. `opponent'` is therefore a one-factor that has `j=opponent[i]` as required. □

Therefore, for either BSRR or for cases where the home/away pattern for a time slot is fixed (creating a bipartition between teams that need to be home versus those that need to be away), one-factor propagation can be replaced by all-different propagation.

5 Computational Tests

If there were no further requirements on the schedules, creating a SRR schedule would be straightforward. Kirkman (1847) gave a method for creating such a schedule (outlined in [1]) which also shows there is an ordering of the decision variables such that a constraint program would not need to backtrack in assigning variables (provided arc-consistent approaches to the all-different constraint is used).

Most league schedules have a number of additional constraints, however. A few of the most common are:

- Fixed games. A set of games are fixed to occur in certain slots.
- Prohibited games. A set of games are fixed not to occur in certain slots.
- Home/Away restrictions. Each team has a home venue, and each game must be assigned to a venue. There are additional constraints on such things as the permitted number of consecutive home or away games.

It might seem that adding constraints would make the problem easier, since it reduces the possible search space. In fact, adding fixed games (or, equivalently, prohibiting games) can make the relatively easy problem of finding a schedule become an NP-complete problem. This has been shown by Colbourn [5] for the bipartite SRR case, where a schedule is equivalent to a Latin Square, and by Easton for the general SRR case. So it is clear that there may be very difficult instances of these restricted problems.

There may also be an objective function to be optimized. For instance, there may be an estimate c_{ijt} for the number of people who would attend a game between i and j during time slot t. Can we maximize the total number of people who attend games during the tournament?

All the testing in this paper was done using ILOG's OPL Studio version 3.5 ([13]) running under Windows XP on a 1.8Gz Pentium IV processor computer with 512Mb of memory.

5.1 Tightly Constrained Round Robin Tournaments

For our first test, we use a data set from HMT, called "Tightly Constrained Round Robin Tournaments". For these instances, there are random forbidden opponents, with a sufficient number to lead to instances with very few or no feasible schedules.

For this test, we compare two codes: Basic-IP and the all-different constraint program. Our codes were implemented within the OPL system, version 3.5 and used default branching strategies for the integer program and default search strategies for the constraint program.

In addition, we repeat the computational results of HMT for their arc-consistent one-factor method. To offset different machine capabilities, we divided their computation times by 4.5 to represent a rough approximation of the difference between their 400Mz machine and our 1.8Gz machine.

For each instance, we give the number of failures (F) in the search tree (for the constraint programs) or number of nodes (N) in the search tree (for the integer program) as well as the computation time in seconds. In the following the instances that end in "yes" are feasible, though generally with a small number of solutions; those that end in "no" are infeasible. For feasible problems, the timing for HMT includes work needed to find all solutions, while those for the IP and the all-different CP only find the first solution. This suggests that the "no" instances, where the codes perform the same task, is the fairer comparison.

Problem	n	all-different		Basic-IP		HMT	
		F	T	N	T	F	T
s_6_yes	6	5	0.00	0	0.01	4	0.01
s_8_yes	8	17	0.01	4	0.04	10	0.04
s_10_yes	10	4	0.02	1	0.09	1	0.02
s_12_yes	12	376	0.41	57	0.46	179	1.39
s_14_yes	14	862	1.24	276	5.54	527	4.53
s_6_no	6	3	0.00	0	0.01	4	0.00
s_8_no	8	11	0.01	10	0.04	6	0.01
s_10_no	10	23	0.02	4	0.10	6	0.03
s_12_no	12	24	0.07	0	0.14	25	0.17
s_14_no	14	135	0.23	50	1.02	69	0.56
s_16_no	16	79	0.30	0	0.39	86	1.19
s_18_no	18	43	0.32	0	0.42	30	0.50
s_20_no	20	696.30	5.47	0	0.78	254	5.11

Table 1. Benchmarks on Tightly Constrained SRR

Basic-IP is competitive with the constraint programming approaches, and can do markedly better in proving infeasibilty (as in `s_20_no`).

This table might lead to the conclusion that these tightly constrained SRR problems are relatively easy to solve in this size range. That is not the case. The instance `s_16_no` is actually just one of a series of instances, corresponding to varying numbers of prohibited games. The instance begins with 1192 prohibited games. The timing above corresponds to prohibiting all but the final 7 of the prohibited games. We can create new instances by varying the number of prohibited games. The instances remain infeasible through prohibiting all but the final 54 games, and which point the instance becomes feasible.

The computational effort for these instances varies tremendously for the Basic-IP and the all-different approaches. Basic-IP does poorly for a broad range of prohibitions (the good behavior above is an anomoly). For instance, the computation time for the feasible instance that is created by prohibiting all but the final 80 games is more than 9000 seconds. The all-different constraint program never does that poorly but can still take more than 300 seconds for various instances. Clearly many of these instances are difficult for our codes even for these relatively small sized instances.

5.2 Divisional Schedules

There is a natural set of restrictions that are extremely difficult for the constraint-based formulations but are solved much more quickly by the integer programs. Many leagues are divided into two or more divisions. In such cases, some leagues like to begin by playing games between divisions and finish the schedule with games between divisional opponents. For two equally sized divisions, this approach works fine if n is divisible by 4, but does not work well for cases where $n = 2 \bmod 4$. In that case, divisions have an odd number of teams, so no compact round robin schedule is possible with only divisional play at the end.

We can create difficult instances by fixing a large number of games. Suppose there are n teams, with $n = 2 \bmod 4$, numbered 0 up to $n-1$. Divide the teams into two groups $X = [0..n/2-1]$ and $Y = [n/2..n-1]$. For the first $n/2-1$ slots, play X and Y as a bipartite tournament, leaving one game between X and Y unplayed for each team. Then, in slot $n/2$ fix two of the remaining games between X and Y. For six teams, the schedule might be:

	Team					
Slot	0	1	2	3	4	5
1	3	4	5	0	1	2
2	4	5	3	2	0	1
3	5	3		1		0
4						
5						

With these fixtures, the schedule is infeasible. In the above example, teams 2 and 4 need to play in slot 3, and then 0, 1, and 2 need to play a round robin among themselves in the remaining two slots, which is impossible.

The size 6 example is easy for any approach; things get more interesting for larger problems, as shown in Table 2. In this table, the dashes mean that no proof of infeasibility was found in half an hour of computation time (1800 seconds).

For the Strong-IP, the only needed constraints are for the odd sets associated with each division in each time period. That is sufficient for the linear relaxation to be infeasible for every n, which gives the near-constant computation times. For the one-factor method of HMT, there is no immediate proof of infeasibility for $n \geq 10$ by domain reduction, so at least some branching is needed (and we believe the amount of work will be significant).

5.3 Maximum Value Schedules

As a final test for SRR, we randomly generated values for each game in each slot and tried to find the maximum value schedule. For an instance of size n, we independently generated a value uniformly among the integers $1 \ldots n^2$ for

Size	all-different		Basic-IP		Strong-IP	
	F	T	N	T	N	T
10	116	0.20	393	0.19	0	.20
14	—	—	—	—	0	.34
18	—	—	—	—	0	.32
22	—	—	—	—	0	.38

Table 2. Divisional Schedules (30 minute time limit)

each game (i, j, t). There were no other restrictions on the schedule. We also generated bipartite versions of these problems (denoted b in the tables below).

For these sorts of optimizations, the search strategy is critical for constraint programming. The strategy used was to set the variables slot by slot, beginning with the first slot. The opponents for each team are ordered in decreasing order by value, and high value opponents are tried first.

The results are shown in Table 3. Not surprisingly, the integer programming based approach does much better at this test. In order for constraint programming to be competitive in this test, some sort of cost-based domain reduction would have to be done. Note that even the bipartite problems are difficult for constraint programming despite the arc-consistent propagation we do through the all-different constraint. HMT's `one-factor` constraint is no stronger in the bipartite case.

Size	all-different		Basic-IP	
	F	T	N	T
8	84962	5.33	0	.03
10	—	—	66	.29
12	—	—	402	3.59
14	—	—	7263	133.03
8b	1458	0.04	0	0.02
10b	3832	0.36	0	0.04
12b	4800172	216.75	0	0.09
14b	—	—	0	0.10

Table 3. Maximum Value Schedules (30 minute timelimit)

6 Home/Away Pattern Restrictions

The final set of constraints we would like to consider is extremely important in practice. For some leagues, every team has a home venue and every game is played at the home venue of one of the two teams competing. In this situation, there are often constraints on the home and away patterns of these teams. These constraints might include

- Restrictions that a particular team be home (or away) in a particular slot.
- Limitations on the number of consecutive home (or away) games a team may play.
- Requirements on the number of home games that must appear in some subset of the slots. For instance, a team might want to be at home at least half of the weekend games, or half of the games during the summer.
- Restrictions on pairs of slots. For instance, a if a team begins with an away game, the final game of the tournament might be required to be a home game.

There has been much work on scheduling with home/away patterns. Much of this work has concentrated on multiple phase approaches, where first the home/away pattern is fixed, and then the games are chosen consistent with this pattern (see, for example, [6, 7, 23, 20, 15, 11]. Alternatively, some work has reversed the process where first the games are chosen and then the home/away pattern chosen ([19, 24]). While these approaches are often very successful, there are cases where they do not work very well. For instance, depending on the restrictions on home/away patterns, there can be a huge number of feasible patterns, and a correspondingly large number of basic match schedules (see [23]). Enumerating and searching through all of them can be a computationally prohibitive task.

Is it possible to have one model that contains both game assignment and home/away pattern decisions? Conceptually, the models are straightforward to formulate. We consider a *double round robin* (DRR) tournament where every team plays every other team twice, once at home and once away.

For the integer program, we reinterpret the x_{ijt} variables to mean that i plays at j during slot t. We also create auxilliary variables h_{it} which is 1 if i is home in slot t and 0 otherwise. Clearly $h_{it} = \sum_j x_{jit}$.

For the constraint program, there are a number of possible formulations. For this test, in order to maximize the effect of the `all-different constraint`, we used the variables `plays[i,j]` to be the slot number in which i plays at j. This gives the basic formulation of

```
forall (i in Teams)
    all-different(all (j in Teams) plays[i,j],
                  all (j in Teams) plays[j,i]);
```

(For notational convenience, we actually created a n by 2n array of variables with `plays[i,j]` for `j <=n` giving the slot where i is at home to j and for `j>n` giving the slot where i is away to $j - n$, but will continue the exposition with the original variables).

Our home/away variables are given by

```
forall (i, j in Teams)
   home[i,plays[j,i]] = 1;
   home[i,plays[i,j]] = 0;
```

For both the integer and constraint programming approaches, some schedule requirements are easy to formulate with these variables. Fixing teams to be at home (or away) at a particular time is simply a matter of fixing the h (or `home`) variable to take on the appropriate value. Fixing the number of home games in a subset of slots is simply a linear constraint on the sum of the home variables. Putting an upper bound on the number of consecutive home games can be done as follows: if no more than k consecutive home games are permitted, then for every i and t, add a constraint

```
sum(t1 in Slots: t>=t & t<=t+k) home[i,t1] <= k
```

A similar restriction on consecutive away games can be done by using `1-home[i,t1]`.

OPL has a stronger way of handling these constraints: the `sequence` constraint allows for the explicit bounding of the number of times a value can appear in a subsequence of an array. We add to the constraint program the redundant constraints that half the teams must be at home in every slot (without them, the constraint program works very poorly).

Our first test is to simply determine whether our programs can find schedules in the absence of additional constraints. It was shown in HMT that constraint programs can find unrestricted schedules quickly for more than 20 teams (and more than 40 teams with their one-factor improvements). How does adding home and away requirements affect that?

In the following table, we give the time to find one schedule with n teams and an upper bound of k consecutive home or away games. For $k = 1$, there is no feasible schedule, so the time given is the time to prove infeasibility. With just one exception, the constraint programming approach did much better, though the integer program was able to generate solutions in a reasonable amount of time.

The entry for the constraint program for $n = 12$, $k = 3$ is not a misprint: despite the ease at which the constraint program solved the other instances, the search went poorly in this case, and no solution was found within one-half hour. This suggests that either an improved search procedure is needed (we simply instantiated the `play` variable before the `home` variable team-by-team) or a stronger propagation algorithm is needed to ensure consistency in computation time. Still, it is clear that constraint programming is by far superior for this type of instance.

To move closer to to the types of schedules needed in practice, we add a constraint that there cannot be any length-one home stands or road trips. This is done by adding constraints of the form

```
home[i,t] <= home[i,t-1]+home[i,t+1];
(1-home[i,t]) <= (1-home[i,t-1])+(1-home[i,t+1]);
```

with the obvious changes for the beginning and end of the schedule. This makes the $k = 2$ instances infeasible.

		Integer Program		Constraint Program	
n	k	N	T	F	T
8	1	4	1.04	40	0.05
8	2	7	1.41	6	0.05
8	3	4	1.04	21	0.04
8	4	0	0.56	4	0.02
10	1	6	8.82	40	0.01
10	2	4	5.92	199	0.24
10	3	1	2.87	462	0.44
10	4	6	3.72	1141	0.98
12	1	6	24.84	220	0.54
12	2	4	17.29	2	0.84
12	3	4	15.11	—	—
12	4	17	32.42	0	0.12
14	1	2	59.71	312	1.21
14	2	9	70.10	11	0.20
14	3	11	82.34	3	0.18
14	4	20	169.42	2	0.19
16	1	2	163.52	420	2.48
16	2	35	604.86	184	0.74
16	3	—	—	197	.32
16	4	124	1557.02	1	.03
18	1	2	669.14	544	4.82
18	2	28	892.64	227	1.16
18	3	—	—	9	.04
18	4	—	—	1	.05

Table 4. Length Constrained H/A Schedules (30 minute limit)

The results are shown in Figure 5. Clearly this approach to limiting the home/away pattern is not consistent with the rest of the constraint programming model: constraint programming is unable to find any feasible solutions with half an hour. The integer programs are slow, but do find solutions. Given the success the multiple-phase approaches have with instances like this, it is clear that a smarter search rule (mimicking the multiple phase approach) or a better propagation rule should have significant effect on the constraint program.

		Integer Program		Constraint Program	
n	k	N	T	F	T
8	2	1427	21.42	2166	0.99
8	3	513	60.12	—	—
8	4	750	83.86	—	—
10	2	115	111.09	42768	26.18
10	3	1354	921.6	—	—
10	4	2214	1290.38	—	—

Table 5. Length Constrained H/A Schedules, No Singles (30 minute limit)

Finally, we added values for matchups on particular days, generating random values in the range $1 \dots n^2$ for each pairing in each slot. Unfortunately, neither code at this stage can solve even the $n = 8$, $k = 3$ instance with a no-singleton constraint within 30 minutes. The results in the table are without the no-singleton constraint.

		Integer Program		Constraint Program	
n	k	N	T	F	T
8	3	1516	22.73	—	—
8	4	77	0.92	—	—
10	3	—	—	—	—
10	4	15268	594.70	—	—

Table 6. Length Constrained H/A Schedules, Maximum Value (30 minute limit)

One final set of requirements we have not yet included has to do with travel requirements on teams. For leagues like MLB where travel a concern, it is important to minimize the travel distances of each team. Unfortunately, the direct formulation of this does not lead to solvable models. More complicated models involving different variables seems to be needed [10].

We can, within the models given, preclude terrible travel by including constraints that require trips from, say, the east coast to the west coast to include at least two west coast teams before returning. Such constraints are similar to (for both formulation and computation) the constraints that preclude length-one homestands and roadtrips.

7 Conclusions

We have shown that round-robin schedules with constraints of practical interest can be modeled by both constraint and integer programming techniques. The constraint programs were often faster except when there was an objective function, or in certain infeasible cases where the propagation was not strong enough to recognize infeasibiilty.

Returning to the Major League Baseball example, once the problem has been divided into smaller pieces, it is clear that IP/CP approaches are reasonable methods to solve the sections. Computation times are low enough to allow for multiple iterations of each section. In these iterations, constraints and objectives can be modified to push the process towards a good overall schedule.

In the course of this study, a number of gaps in current knowledge have been identified, and these make interesting future research directions.

- Is it worth adding constraints via the Strong-IP formulation? Henz, Müller, and Thiel [12] show that stronger propagation is generally a good idea for constraint programs. Is it also true that stronger relaxations are good for these integer programs?
- How can costs be better handled for the constraint programs? Handling costs is an active issue in the constraint programming community, and round-robin scheduling makes a good test-bed for these approaches.
- Clearly it would be good to include the strong propagation of HMT to the home/away models. Is there stronger propagation available combining the opponents with the home/away structures? Are there additional constraints that can be added to the integer programs? Is there a better way of handling home/away models which does not require a multi-phase approach?
- Can the integer programming and constraint programming approaches be usefully combined for these problems?

Round-robin scheduling makes an interesting test-bed for exploring algorithmic issues in combinatorial optimization. Success in this scheduling also provides the building blocks for scheduling of real-world sports leagues.

References

1. Anderson, I. 1997. *Combinatorial Designs and Tournaments*, Oxford University Press.
2. Ball, B.C. and D.B. Webster. 1977. "Optimal scheduling for even-numbered team athletic conferences", AIIE Transactions 9, 161-169.
3. Cain, W.O., Jr. 1977. "A computer assisted heuristic approach used to schedule the major league baseball clubs", in Optimal Strategies in Sports, S.P. Ladany and R.E. Machol (eds.), North Holland, Amsterdam, 32-41.
4. Campbell, R.T. and D.-S. Chen. 1976. "A minimum distance basketball scheduling Problem", in Management Science in Sports, R.E. Machol, S.P. Ladany, and D.G. Morrison (eds.), North-Holland, Amsterdam, 15-25.

5. Colbourn, C.J. 1983. "Embedding partial Steiner triple Systems is NP-complete", Journal of Combinatorial Theory, Series A, 35, 100-105.
6. de Werra, D. 1980. "Geography, games, and graphs", Discrete Applied Mathematics 2, 327-337.
7. de Werra, D. 1988. "Some models of graphs for scheduling sports competitions", Discrete Applied Mathematics 21, 47-65.
8. Easton, K.K. 2002. *Using Integer Programming and Constraint Programming to Solve Sports Scheduling Problems*, doctoral dissertation, Georgia Institute of Technology.
9. Edmonds, J. 1965. "Maximum matching and a polyhedron with (0,1) vertices", *Journal of Research of the National Bureau of Standards Section B*, 69B, 125-130.
10. Easton, K.K., G.L. Nemhauser, M.A. Trick. 2002. Solving the Traveling Tournament Problem: A Combinted Integer Programming and Constraint Programming Approach, PATAT IV, Gent, Belguim.
11. Henz, M. 2001. "Scheduling a Major College Basketball Conference: Revisted", Operations Research, 49, 163-168.
12. Henz, M., T. Müller, and S. Thiel. 2003. "Global Constraints for Round Robin Tournament Scheduling", to appear, European Journal of Operational Research.
13. ILOG. 2000. "ILOG OPL Studio", User's Manual and Program Guide.
14. Marriott, K. and P.J. Stuckey. 1998 *Programming with Constraints: An Introduction*, MIT Press.
15. Nemhauser, G.L. and M.A. Trick. 1998. "Scheduling a Major College Basketball Conference", Operations Research, 46, 1-8.
16. Padberg, M.W. and M.R. Rao. 1982. "Odd minimum cut-sets and b-matchings", Mathematics of Operations Research, 7, 67-80.
17. Régin, J.-C., 1994. "A filtering algorithm for constraints of difference in CSPs", *Proceedings of the AAA 12th National Conference on Artificial Intelligence*, 362–367.
18. Régin, J.-C., 1999. "The symmetric alldiff constraint", in T. Dean (ed) *Proceedins of the International Joint Conference on Artificial Intelligence*, 1, 420-425.
19. Régin, J.-C. 1999. "Minimization of the Number of Breaks in Sports Scheduling Problems using Constraint Programming", DIMACS Workshop on Constraint Programming and Large Scale Discrete Optimization.
20. Russell, R.A. and J.M Leung. 1994. "Devising a cost effective schedule for a baseball league", Operations Research 42, 614-625.
21. Schaerf, A. 1999. "Scheduling Sport Tournaments using Constraint Logic Programming", Constraints 4, 43-65.
22. Schreuder, J.A.M. 1980. "Constructing timetables for sport competitions", Mathematical Programming Study, 13, 58-67.
23. Schreuder, J.A.M. 1992. "Combinatorial aspects of construction of competition Dutch Professional Football Leagues", Discrete Applied Mathematics 35, 301-312.
24. Trick, M.A. 2001. "A schedule-then-break approach to sports timetabling", in E. Burke and W. Erben (eds) *Practice and Theory of Automated Timetabling III*, LNCS 2079, Springer.
25. Van Hentenryck, P. 1999. *The OPL Optimization Programming Language*, MIT Press.