A Schedule-then-Break Approach to Sports Timetabling

Michael A. Trick

GSIA, Carnegie Mellon, Pittsburgh PA 15213, USA
trick@cmu.edu
http://mat.gsia.cmu.edu

Abstract. Sports timetabling algorithms need to be able to handle a wide variety of requirements. We present a two-phase method where the first phase is to schedule the teams ignoring any home and away requirements and the second phase is to assign the home and away teams. This approach is appropriate when there are requirements on the schedule that do not involve the home and away patterns. Examples of this include fixed game assignments and restrictions on the effect carry-overs can have. These requirements can be met early in the process and the best home and away patterns can then be found for the resulting schedules.

1 Introduction

Creating a sports timetable is challenging due to the variety of different needs that must be addressed. This has led to a plethora of alternative approaches. While many of the approaches have a common thread due to their division of the problem into a number of common subproblems, these methods differ in the order in which subproblems are solved and the solution method(s) chosen.

We present results on an alternative approach for solving a standard sports timetabling problem. In this problem, teams must play a round-robin tournament among themselves, with each game played at the home field of one of the competing teams. We assume that the number of teams is even and that during every time period, or slot, every team plays exactly one game. This sort of timetable occurs very often in competitions where each team represents a city or other region and the competition occurs over an extended period of time. This contrasts with the situation where many teams share a single field, or where all teams are brought to a single location for the competition.

This round robin scheduling, or variants thereof, has appeared often in the literature. Examples include minor league baseball (Russell and Leung [17]), college basketball (Ball and Webster [2]; Nemhauser and Trick [12]; Walser [22]; Henz [9]), Australian basketball (de Werra, Jacot-Descombes, and Masson [25]), and Dutch professional football (Schreuder [20]), and has been explored extensively independent of particular sport (de Werra [23, 24]; Schreuder [19]; Henz [8]; Schaerf [18]).

In this paper, we look at an alternative approach to solving these types of problems. In this approach, we divide the problem into two phases. In the first phase (the scheduling phase), we assign games to be played without concern for the home/away decision. Our second phase (the break phase) then assigns home and away teams in such a way as to minimize the occurrence of bad structures (here defined to be consecutive home games or consecutive away games for a team, or breaks). This approach is particularly useful when there are requirements on the schedule that don't involve home/away patterns. An example is to create a schedule where a number of games are fixed a priori.

This approach uses constraint and integer programming to solve the two phases. The scheduling problem is a natural combinatorial design problem. This can easily be extended to allow fixed games and to implement the carry-over restrictions of Russell [16].

The break problem has been studied by Régin in [13, 14]. In this work, we present an integer programming formulation that is able to solve larger instances than the constraint programming method of Régin. Our success is due in part to applying Régin's discoveries in constraint programming to our integer programming formulation.

Section 2 outlines the standard multiphase approach to sports timetabling and gives our general approach to this problem. Section 3 then develops our scheduling phase and Section 4 gives our break phase. We conclude with some open problems.

2 Multiphase Approach to Sports Timetabling

Sports timetabling problems come in two broad types: temporally relaxed and temporally constrained. In a temporally relaxed schedule, the number of days on which games can be played is larger than the minimum number needed. Examples of such timetables include that of (American) National Basketball (Bean and Birge [3]) and National Hockey League schedules (Ferland and Fleurent [7]). In these leagues, teams may play on any day of the week, but would typically only play on 2 or 3 of them in any week. These schedules may look difficult, due to the additional decision of choosing a day on which to play, but many effective heuristic techniques have been developed. Other examples of competitions with temporally relaxed schedules include the scheduling of cricket matches (Armstrong and Willis [1]; Willis and Terrill [26]; Wright [27]).

In a temporally constrained schedule, the length of the schedule is chosen so there is just enough time to play all of the games. While there is no longer a decision on when to play (each team will play as much as possible), there is also no longer the extra flexibility needed to apply the heuristic techniques listed above. Determining a reasonable local search neighborhood seems a difficult question in a temporally constrained problem, though Walser [22] reports success using his general local search methods for integer programs.

A number of researchers have attacked this problem in various ways (Ball and Webster [2]; Henz [8, 9]; Nemhauser and Trick [12]; Russell and Leung [17];

Schaerf [18]; Schreuder [19, 20]; de Werra [23, 24]; de Werra, Jacot-Descombes, and Masson [25]). All of this work can be seen as addressing a number of subproblems. These subproblems include the following:

1) Finding Home-Away Patterns (HAP). Find a set of n strings of H (Home) and A (Away) of length n-1 that corresponds to the home and away sequence of a team.

For six teams, a possible HAP would be

```
1: HAHAH
2: AHAHA
3: HHAAH
4: HAHHA
5: AAHHA
6: AHAAH
```

2) Assign games consistent with the HAP (so if i plays j in a slot, then either i is home and j is away, or the reverse). For the above, we might get the following timetable (+ denotes at home; - is away), to get a Basic Match Schedule (BMS):

```
1: +2 -3 +6 -4 +5
2: -1 +4 -5 +6 -3
3: +6 +1 -4 -5 +2
4: +5 -2 +3 +1 -6
5: -4 -6 +2 +3 -1
6: -3 +5 -1 -2 +4
```

3) Assign teams to the BMS to complete the timetable. If the teams were A,B,C,D,E,F, then we might assign them to 3,6,5,4,2,1 respectively to get the final timetable:

```
F: +E -A +B -D +C
E: -F +D -C +B -A
A: +B +F -D -C +E
D: +C -E +A +F -B
C: -D -B +E +A -F
B: -A +C -F -E +D
```

Alternative approaches to sports timetabling differ in the order in which they solve the subproblems and the method employed to solve each of the subproblems. For instance, Nemhauser and Trick [12] solved the subproblems in the order given above and used integer programming for problems 1 and 2, and complete enumeration for problem 3. Henz [9] then improved on this by using constraint programming for all three steps, where the speedup on problem 3 was enormous. Walser [22] used a heuristic integer programming technique to avoid breaking the problem into three subproblems.

The decision on which order to do the subproblems depends on the importance of the decision being made. In general, the more critical an aspect of a schedule, the more important it is to fix that decision early in the process. The above ordering makes the form of the Home-away patterns most important. Leagues with other requirements may need to consider alternative orderings.

We will consider the case where the game assignment issue in step 2 is critical, even ignoring the home and away aspects. For instance, there might be a number of pre- assigned games that must be played in particular slots. Examples where this occurs include required television matchups and required "traditional matchup" requirements. Both of these occurred in the creation of the Atlantic Coast Conference basketball schedule of Nemhauser and Trick [12]. While the requirements in the year given in that paper were not extensive, latter years added many more required match opponents, which made the original multiple phase approach presented less appealing.

Another reason to consider step 2 (even before home/away assignment) is the work of Russell [16]. Russell looked at the issue of carry-over effects: if A plays B in round 1, then plays C in round 2, then C gets a carry-over effect from B. If B is a very difficult team, then C may gain an advantage. Russell showed that if the number of teams is a power of 2, then the advantages can be spread out evenly among the teams, and he gave a construction in this case. For other sizes, he gave a heuristic solution to minimize the variance of the carry-over effect. These give schedules to which home and away decisions would then be applied.

Our approach is therefore to first schedule the teams and then find the home team for each game. This is the following two-phase approach:

1) Find a schedule of the teams (ignoring home and away) that is consistent with the required match opponents or other requirements on the schedule (schedule phase).

2) Assign the home team to each game to give good home/away patterns (break phase).

The definition of good home/away patterns is very league-dependent. Wallis [21] looked at this problem in the context of creating a schedule so that the teams are paired with one being at home and one away during each period. Schreuder [19] and de Werra [23, 24] addressed this in the context of minimizing the breaks in a timetable. In these models, the ideal pattern is alternating homes and aways (this is a very reasonable requirement for many real leagues). Any deviation from this, in the form of two consecutive aways or two consecutive homes, is a break. Schreuder and de Werra developed a number of structural conditions and bounds on the number of breaks. This work combined the schedule and the break phase to give schedules with the minimum number of breaks.

Régin [13, 14] looked at the break phase for arbitrary schedules and gave a constraint programming approach to the problem of minimizing the number of breaks. We will continue this direction by adopting a definition of "good home/away patterns" to mean the set of patterns that minimizes the number of breaks.

The next two sections look at each phase of this approach in turn.

3 Schedule Phase

The schedule phase might be solved in a number of ways, depending on the exact requirements on the schedule. In the absence of constructive techniques, like those of Russell [16], we need a computational method for finding schedules. Ideally, such a method would be flexible enough to allow alternative evaluation rules, and allow such extensions as fixing particular games. We will concentrate on the issue of fixed games and conclude with some comments on the carry-over model.

Given n teams (labeled 1..n), and a series of triplets (i,j,t), find a round-robin schedule for the teams such that i plays j in slot t. For six teams, the input might look like:

```
Slot 1 2 3 4 5
A:   B F   C
B:   A   F
C:   D   E A
D:   C E
E:   F D C
F:   E A B
```

One output would be:

```
Slot 1 2 3 4 5
A:   B F D C E
B:   A C F E D
C:   D B E A F
D:   C E A F B
E:   F D C B A
F:   E A B D C
```

This problem is an example of a "completion problem" in combinatorial design. Colbourn [6] showed that many of these problems, including the one we address, are NP-complete (de Werra [24] gives some instances related to sports scheduling that can be solved quickly). Our goal is to create a reasonable computational method, rather than solve the problem for extremely large instances. Therefore, we explore two alternative constraint programming approaches for generating solutions to these problems.

We begin with the case that there are no required game placements. In this case, we are only trying to generate tournament schedules, a problem closely related to finding latin squares. Of course, many construction techniques are known for finding a schedule, but these typically find only one or a small number of schedules, so are unsuitable for our purposes.

Our first formulation for this is perhaps the most natural. A variable is set for each (team,time) pair which gives the opposing team in that time slot. Constraints are added to ensure that every team plays every other team and that symmetry holds: if i plays j then j plays i. This formulation can be strengthened by including the constraint that for every time slot, every team plays some opponent. We can also break the symmetry of the formulation by labeling team 1's opponents 2, 3,...,n in order. We call this the Opponent Formulation. The OPL (ILOG [10]) code for this is given in Figure 1.

```
int n=20;
range Teams 1..n;
range Time 0..n-2;
range Opponent 1..n;
var Opponent opponent[Time,Teams];

solve {
   forall (i in Teams) {
      alldifferent(all (t in Time)(opponent[t,i]));
      forall (t in Time) {
       opponent[t,opponent[t,i]] = i;
      opponent[t,i] <> i;
      };
   };
   forall (t in Time) {
      alldifferent(all (i in Teams) (opponent[t,i]));
      opponent[t,1] = t+2;
   };
};
```

Fig. 1. Opponent Formulation

An alternative formulation is to give, for each pair of teams, the slot in which the teams play. Each team must play only one game in a slot, and symmetry can be broken as above. This is the Slot Formulation. The OPL code is in Figure 2.

The formulations act very similarly (and an appropriate search strategy can make them act nearly identically), but some constraints are easier to write in one model than the other. Each model can generate a 20-team schedule in less than 1 second and generate 500 20-team schedules in around a minute. OPL has the capability of using different propagation algorithms for the "alldifferent" constraint: the strongest setting of "onDomain" gave the best results for these tests.

Rosa and Wallis [15] explored the concept of "premature sets": assignments of games to slots in such a way that the schedule could not be completed. They showed that such premature sets could occur and placed bounds on the size of the sets. Perhaps surprisingly, neither formulation for this problem had any difficulty with these premature sets: no backtracking was needed for any of the instances up to size 20. This is not true for larger instances.

The behavior of these formulations is not changed significantly by the addition of game requirements (of course, the problem is difficult, so there must exist

```
int n=20;
range Teams 1..n;
range Slots 0..n-1;
var Slots slot[Teams,Teams];

solve {
   forall (i in Teams) {
      alldifferent(all (j in Teams)(slot[i][j]));
      alldifferent(all (j in Teams)(slot[j][i]));
      slot[i,i] = 0;
      forall (j in Teams) {
         slot[i,j] = slot[j][i];
   slot[1,j] = j-2;
      };
   };
};
```

Fig. 2. Slot Formulation

difficult instances, but they do not seem to occur in the 10-20 team instances of interest). Empirically, it seems that adding game requirements makes this phase solve more quickly. Other types of constraints may be more difficult to handle, but forced pairings seem easy.

It should be noted that the Opponent Formulation could be modified to find the balanced- carry-over schedules of Russell [17]. Simply adding the constraint:

```
alldifferent(all (t in time: t>0)
    (opponent[t-1,opponent[t,i]]));
```

prevents a team from having more than one carry-over effect on team i. The resulting formulation solves quickly for 8 teams, and very slowly (on the order of a day) for 16 teams. Russell (1980) also addresses the question of minimizing the variance of the carry-over effects (equivalently the sum of squared carry-over counts). Again this is simply the addition of a constraint to either scheduling formulation. For six teams, he found a schedule with a value of 60, and we can confirm that is indeed optimal. For ten teams, his value of 138 is suboptimal: there exists a schedule of value 122. The computational time for a straightforward modification of our formulations are very high (it took one day to find the ten team solution), so this is an opportunity for further work.

4 Break Phase

The second phase of our approach is to assign the home team (and away team) to each game in a schedule to create a timetable. We wish to assign home/away teams in such a way as to minimize the number of breaks (consecutive home games or consecutive away games). This problem turns out to be much more difficult than the first phase.

Régin [13, 14] has examined this problem in detail using constraint programming. Using a natural formulation (with 0-1 variables) and extensive symmetry breaking and dominance rules, his best results take 5.2 seconds for a 16 team instance, 80 seconds for an 18 team instance, and 5603 seconds for a 20 team instance (Régin's timings are on a 200Mz Pentium computer, so the "normalized" timings for a 266 Mz machine would be 3.9 seconds, 60 seconds, and 4213 seconds respectively).

We present an integer programming model for this problem that is at least competitive with Régin's constraint programming model. While the model is not difficult, it has a number of pieces, all of which are needed in order to have the model work effectively.

The variables for this formulation could be chosen in many ways. We begin with three major variables:

```
start[i]: 1 if team i starts at home, and 0 otherwise.
to_home[i,t]:1 if team i goes home after slot t, and 0 otherwise
to_away[i,t]:1 if team i goes away after slot t, and 0 otherwise
```

From these variables, it is easy to determine if team i is home in slot t or not:

```
at_home[i,t]= start[i]+sum(t1<t)(to_home[i,t1]-to_away[i,t1])
```

is 1 if team i is at home in time t, and 0 if team i is away in time t.

We limit the at_home variables to be 0 or 1. The at_home variables are not needed in the formulation (we can replace them with the identity above), but they make the exposition cleaner.

The primary constraint is that for every pair of teams i, j, if they play in slot t then

```
at_home[i,t]+at_home[j,t] = 1
```

We also need a constraint that for every team i and time t, the team cannot go both home and away:

```
to_home[i,t]+to_away[i,t] <= 1
```

This formulation is a sufficient integer program. Unfortunately, it works extremely poorly: even for an 8 team example, the program takes 81 seconds, while a 10 team example would not solve in 1200 seconds.

To improve this formulation, we begin by breaking symmetry as Régin did: we assume that team 1 begins at home. Since clearly any schedule can be "flipped", reversing home and away, such an assignment does not affect the optimal solution. This has only a marginal effect, decreasing the 8 team time to 75 seconds without allowing us to solve the 10 team example (in 1200 seconds).

We then try to strengthen the constraints to avoid unwanted fractional values that slow down the branch and bound search. We can strengthen the use of to_home and to_away by noting that the requirement on at_home does not prohibit setting both to_home[i,t] and to_away[i,t] to each be 0.5 no matter what at_home[i,t] is. We can therefore strengthen the formulation by adding the constraints:

```
at_home[i,t]+to_home[i,t] <= 1;
at_home[i,t]-to_away[i,t] >= 0;
```

for all i and t. This formulation is much better, solving the 8 team example in 4 seconds and the 10 team example in 75 seconds.

To improve performance, we need to add additional constraints that force breaks to be taken when the teams need them. Consider three teams A, B, and C such that A plays B in slot 1, B plays C in slot 2, and A plays C in slot 3. Then, it is easy to see that either A has a break after slots 1 or 2, or B has a break after slot 1, or C has a break after slot 2. To see this, suppose A is home to B in slot 1. If there are no breaks, then B is home to C in slot 2, so C is home in slot 3. But A is also home in slot 3! Therefore, there must be a break involving one of the three teams. This generalizes to any three teams in any three slots: if we have teams i, j, and k who play in slots slot[i,j]<slot[j,k]<slot[i,k], then either

- i has a break between slot[i,j] and slot[i,k], or
- j has a break between slot[i,j] and slot[j,k], or
- k has a break between slot[j,k] and slot[i,k]

This triangle constraint is very powerful because it links teams and forces one or more of them to take a break when the fractional solution might not do so. The resulting OPL code for this formulation is shown in Figure 3.

We can improve on this formulation in a number of different ways. In general, we will try to exploit knowledge of what solutions must look like in order to avoid unnecessary branching. Two improvements are

- Breaks come in pairs (if one team goes AA, then another team must go HH). Therefore, once the gap in the branch and bound tree is less than 2, the search can stop.
- In any slot, there are either no breaks or at least two breaks.

We can branch on auxiliary variables that model this latter requirement. In the tests below, we simply created two subproblems: the first with no breaks in the first period, and the second with at least two. This alone was a great improvement over just branching on the natural variables. The instances we were looking at now become trivial: the 8 team example solves in .4 seconds while the 10 team example takes 3.5 seconds. Table 1 shows values for larger instances:

It should be noted that this is not a serious computational test: it is only on the size 20 instance that we are even solving the same instance as Régin. On the smaller problems, the differences are well within the range of variation

Size	Régin	IP
16	4	22
18	60	43
20	4213	1092
22		2293

Table 1. Sample Computation Times

across individual instances. It is clear, however, that the IP approach is at least comparable to the constraint programming approach of Régin, and seems to be scaling up perhaps a little better.

5 Extensions and Conclusions

We have presented an alternative approach to creating sports timetables in cases where a fixed game assignment is a critical feature. We presented effective solution techniques for the two subproblems and showed how they could be used for leagues with up to approximately 20 teams.

There are a number of extensions that would make this work more applicable to real sports leagues. Perhaps foremost would be extending this work to double round-robin tournaments. Such a schedule is often played in real leagues, and leads to the additional complication that of the two games between a pair of teams, each team must be at home for one of them. This makes the problem much harder to solve.

Another approach would be to apply a more realistic measure of good home/away patterns. While breaks are important, some breaks are worse than others. For instance, a sequence like AAA is often much worse than AAHAA, although each have 2 breaks. And many leagues, including many US college basketball conferences and Major League Baseball actually prefer to go 2 games at home or 2 away before breaking. How can sequences like AAHHAAHH be encouraged in this model?

Sports scheduling provides a rich field for creating and testing constraint and integer programming formulations. The Break Phase solution presented was helped by the constraint programming efforts of Régin. Perhaps the biggest question of this work is to determine how integer programming and constraint programming can work together to solve these problems.

```
int n= ...;
range Teams 1..n;
range Time [1..n-1];
range TweenTime [1..n-2];
range Binary [0..1];
range Slot [0..n-1];
Slot slot[Teams,Teams] = ...;//slot[i,j] is the slot when i plays j
var Binary to_away[Teams,TweenTime];
var Binary to_home[Teams,TweenTime];
var Binary start[Teams];

range Obj [0..n*n];
var Obj obj;

maximize
   obj
subject to {
//Minimizing breaks is the same as maximizing non-breaks
   obj = sum (i in Teams, t in TweenTime)(to_away[i,t]+to_home[i,t]);
   start[1] = 1;
   forall (i in Teams, t in TweenTime ) {
      to_away[i,t]+to_home[i,t] <= 1;
   };
   forall (i in Teams, t in [2..n-1]) {
      0<= start[i]+sum(t1 in [1..t-2]) (to_home[i,t1]-
                                 to_away[i,t1])+to_home[i,t-1] <= 1;
      0<= start[i]+sum(t1 in [1..t-2]) (to_home[i,t1]-to_away[i,t1])
                                 -to_away[i,t-1] <= 1;
   };
   forall (i in Teams, t in [2..n-1]) {
      0<= start[i]+sum(t1 in [1..t-1]) (to_home[i,t1]-to_away[i,t1])
                                          <= 1;
   };
   forall (ordered i,j in Teams) {
      start[i]+start[j] + sum(t in [1..slot[i,j]-1])
                              (to_home[i,t]-to_away[i,t])
      + sum (t in [1..slot[i,j]-1]) (to_home[j,t]-to_away[j,t]) = 1;
   };
   forall (i,j,k in Teams : (i<>k) & (j<>k) & (i<>j) &
                     (slot[i,j]<slot[j,k]) & (slot[j,k]<slot[i,k])) {
      sum(t in [slot[i,j]..slot[i,k]-1]) (to_away[i,t]+to_home[i,t]) +
      sum(t in [slot[i,j]..slot[j,k]-1]) (to_away[j,t]+to_home[j,t]) +
      sum(t in [slot[j,k]..slot[i,k]-1]) (to_away[k,t]+to_home[k,t])
      <= (slot[i,k]-slot[i,j])+
              (slot[j,k]-slot[i,j])+
              (slot[i,k]-slot[j,k]) -1;
   };
};
```

Fig. 3. Break Formulation

References

1. Armstrong, J. and R.J. Willis. 1993. "Scheduling the cricket world cup: a case study", Journal of the Operational Research Society 44, 1067-1072.
2. Ball, B.C. and D.B. Webster. 1977. "Optimal scheduling for even-numbered team athletic conferences", AIIE Transactions 9, 161-169.
3. Bean, J.C. and J.R. Birge. 1980. "Reducing traveling costs and player fatigue in the National Basketball Association", Interfaces 10, 98-102.
4. 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.
5. 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.
6. Colbourn, C.J. 1983. "Embedding partial Steiner triple Systems is NP-complete", Journal of Combinatorial Theory, Series A, 35, 100-105.
7. Ferland, J.A. and C. Fleurent. 1991. "Computer aided scheduling for a sports league", INFOR 29, 14-24.
8. Henz, M. 1999. "Constraint-based Round Robin Tournament Planning", Proceedings of the 1999 International Conference on Logic Programming, Las Cruces, NM.
9. Henz, M. 2000. "Scheduling a Major College Basketball Conference: Revisted", Operations Research, to appear.
10. ILOG. 2000. "ILOG OPL Studio", User's Manual and Program Guide.
11. Lovasz, L. and M.D. Plummer. 1986. Matching Theory, North Holland, Amsterdam.
12. Nemhauser, G.L. and M.A. Trick. 1998. "Scheduling a Major College Basketball Conference", Operations Research, 46, 1-8.
13. 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.
14. Régin, J.-C. 2000. "Modeling with Constraint Programming", Dagstuhl Seminar on Constraint and Integer Programming.
15. Rosa, A. and W.D. Wallis. 1982. "Premature sets of 1-factors or how not to schedule round robin tournaments", Discrete Applied Mathematics, 4, 291-297.
16. Russell, K.G. 1980. "Balancing carry-over effects in round robin tournaments", Biometrika 67, 127-131.
17. Russell, R.A. and J.M Leung. 1994. "Devising a cost effective schedule for a baseball league", Operations Research 42, 614-625.
18. Schaerf, A. 1999. "Scheduling Sport Tournaments using Constraint Logic Programming", Constraints 4, 43-65.
19. Schreuder, J.A.M. 1980. "Constructing timetables for sport competitions", Mathematical Programming Study, 13, 58-67.
20. Schreuder, J.A.M. 1992. "Combinatorial aspects of construction of competition Dutch Professional Football Leagues", Discrete Applied Mathematics 35, 301-312.
21. Wallis, W.D. 1983. "A tournament problem", Journal of the Australian Mathematics Society Series B, 24, 289-291.
22. Walser, J.P. 1999. Integer Optimization by Local Search: A Domain-Independent Approach, Springer Lecture Notes in Artificial Intelligence 1637, Springer, Berlin.
23. de Werra, D. 1980. "Geography, games, and graphs", Discrete Applied Mathematics 2, 327-337.

24. de Werra, D. 1988. "Some models of graphs for scheduling sports competitions", Discrete Applied Mathematics 21, 47-65.
25. de Werra, D., L. Jacot-Descombes, and P. Masson. 1990. "A constrained sports scheduling problem", Discrete Applied Mathematics 26, 41-49.
26. Willis, R.J. and B.J. Terrill. 1994. "Scheduling the Australian state cricket season using simulated annealing", Journal of the Operational Research Society 45, 276-280.
27. Wright, M. 1994. "Timetabling county cricket fixtures using a form of tabu search", Journal of the Operational Research Society 45, 758-770.