Scheduling Sport Tournaments using Constraint Logic Programming

Andrea Schaerf
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy.
Ph: +39 06 4991 8485, fax: +39 06 8530 0849
e-mail: aschaerf@dis.uniroma1.it
www: http://www.dis.uniroma1.it/~aschaerf

Key words: Sport Scheduling, Constraint Logic Programming, Scheduling Applications, Branch & Bound, Local Search

Abstract. We tackle the problem of scheduling the matches of a round robin tournament for a sport league. We formally define the problem, state its computational complexity, and present a solution algorithm using a two-step approach. The first step is the creation of a tournament pattern and is based on known graph-theoretic results. The second one is an assignment problem and it is solved using a constraint-based depth-first branch and bound procedure that assigns actual teams to numbers in the pattern. The procedure is implemented using the finite domain library of the constraint logic programming language ECL^{i}PS^e. Experimental results show that, in practical cases, the optimal solution of the assignment problem (which is not necessarily optimal for the overall problem) can be found in reasonable time, despite the fact that the problem is NP-complete. In addition, a local search procedure has been developed in order to provide, when necessary, an approximate solution in shorter time.

1. Introduction

Many sport leagues (e.g. football, hockey, basketball) face the problem of scheduling the matches of the round robin tournament. The problem consists in assigning matches to rounds in such a way that every team plays with every other one, all teams play every round with a different opponent (either home or away), and various other side constraints are satisfied.

This problem has a straightforward graph-theoretic formulation, and several papers have appeared in the literature concerning the solution of different variants of the problem based on properties of the corresponding graphs (see e.g., de Werra, 1980, de Werra, 1985, de Werra *et al.*, 1990, Schreuder, 1980, Straley, 1983).

In addition, a considerable attention has been devoted to the automated generation of the schedule using computer programs. To this respect, various techniques have been used: heuristics (see e.g., Cain,

1977, Ferland and Fleurent, 1991), *clustering* (Schreuder, 1992), and *tabu search* (Costa, 1995).

We deal with the specific problem of finding a schedule of a round robin tournament for a sport league with various constraints including availability for matches and stadia, like the Dutch "Top League" or the Italian "Serie A" of football (USA: soccer).

We tackle the problem using a two-step approach (as has been already proposed in Schreuder, 1992). The first step regards the generation of a fixed tournament pattern, which can be done in polynomial time using known graph-theoretic results. The second step is the *team assignment* and it involves the solution of a *bipartite graph matching* with side constraints, which turns out to be an NP-complete problem.

We present an exact solution of the team assignment problem based on a depth-first branch and bound technique implemented in the logic programming language ECL^{i}PS^e (ECRC, 1995b) using the *finite domain* library (ECRC, 1995a, Chapter 5). Using a suitable ordering of the selected variables and their values, and thanks to the good pruning capability of the *finite domain* constraint solver, we have been able to find for practical cases the optimal solution of the team assignment problem in a reasonable computation time.

It is worth mentioning that the solution found is not necessarily optimal for the general tournament scheduling problem, unless the tournament pattern is fixed for the problem.

We also provide an approximate solver based on local search and implemented in C++, that allows the user, when necessary, to come up with a solution (even if not optimal) in very short time.

The paper is organized as follows: Section 2 defines the round robin tournament problem. Section 3 describes how the problem can be tackled in a two-step way, dividing it into two smaller subproblems. Section 4 discusses the computational complexity of the overall problem and its subproblems. Section 5 explains the algorithm employed and its implementation. Section 6 shows the experimental results and the performances obtained. Section 7 describes the approximate solution and the interactive features of the system. Finally, in Section 8 related and future work is discussed and some conclusions are drawn.

2. Tournament Scheduling

A league comprises $2n$ teams, and in $2n - 1$ consecutive rounds each team must play with each other. Matches take place at the home stadium of one of the two teams, and home and away games should be alternated as much as possible. We call *break*, after de Werra (1980),

the fact that a team plays two consecutive rounds in the same *location*, where the term location denotes either home or away. The problem is to find a schedule that minimizes the number of breaks and satisfies a number of other side constraints.

Constraints are split into hard (requirements) and soft ones (wishes): Hard constraints must necessarily be satisfied by the solution, soft ones instead can be violated and they contribute to the objective function.

For all the types of constraints defined below, each single constraint can be declared either hard or soft. The soft ones are associated with an integer-valued positive penalty, and the sum of all penalties determines the objective function. The hard ones are strictly enforced during the construction of the solution.

We have two groups of constraints. The first group, that we call *ordinary constraints*, regards general constraints on all teams. The second group of constraints are related to a grouping of the teams based on their strength, and we call them *special constraints*.

2.1. Ordinary constraints

Ordinary constraints we consider are of the following types (see also Schreuder, 1993).

- **Complementarity**: Teams t_1 and t_2 must have *complementary schedules*. The schedules of two teams t_1 and t_2 are called complementary when, for each round r, if t_1 plays home in round r then t_2 plays away in round r, and if t_1 plays away in round r then t_2 plays home in round r.
- **Availability**: Team t must play home (or away) at round r.
- **Mating**: Team t_1 cannot play home (or away, or both) with team t_2 at round r.
- **Triples**: Three teams t_1, t_2, and t_3 cannot be simultaneously in the same location (i.e. two must be in one location and the third in the other location).

Hard complementarity constraints are used if two teams share the same stadium (e.g. the "San Siro" stadium in Milan is shared by Internazionale and A.C. Milan). Soft complementarity instead is used if the stadia of two teams are located close to each other and the clubs want to optimize the use of railways and highways for their supporters (e.g. Feyenoord and Sparta have their stadia in Rotterdam).

Hard availability constraints are used when a stadium is occupied by some other event in a given round (e.g. a team playing in another league

like for Sampdoria in "Serie A" which shares the "Marassi" stadium with Genoa in "Serie B"). Soft availability constraints are used either for commercial aspects (e.g. overlapping with other events), sportive aspects (e.g. clubs promoted from the inferior league play the first game at home), or organizational aspects (e.g. clubs with hooligans among the fans should not be allowed for away games in a round scheduled in a week day).

Hard and soft mating constraints are mostly used for sportive aspects. For example matches between teams of the same city (*derby matches*) should not occur in the first or in the last rounds. Further, teams involved in the European cups should not play with a strong opponent just before the cup matches.

Triples constraints are used for triples of teams which are located closed to each other in one geographic area. The scheduling of three matches simultaneously in that area would overload railways and highways due to traveling supporters.

Soft complementarity constraints are penalized proportionally to the number of rounds in which they are violated. Therefore, their penalty weight is multiplied by the number of times that the two teams play in the same location (which varies from 0 to $2n-2$). For soft complementarity, it is usually requested that the optimal solution satisfies not only a minimality condition, but it also ensures a certain level of fairness. In fact, a solution cannot be acceptable if it optimizes the objective function at the expenses of some specific teams. We improve fairness by adding hard constraints that impose that certain soft constraints are not violated beyond a given extent. To this aim, we also consider constraints of the following kind

– **Fairness**: Teams t_1 and t_2 can be simultaneously in the same location (home or away) for at most m rounds.

Generally, for each soft complementarity constraint we associate a hard fairness constraint that ensures that the complementarity is violated at most m times in the season.

Ordinary constraints have been already introduced in (Schreuder, 1993), even though Schreuder does not consider all combinations of hard and soft constraints.

2.2. Special constraints

The definition of the second group of constraints presupposes some prior notions: We call *top teams* the members of a subset of the teams composed by the strongest teams, which require some special treatment. We call *top match* a match between two top teams. We call

distance of two matches the number of rounds between the rounds in which they take place.

The special constraints are all hard constraints, and they are the following ones (which are considered also in Schreuder, 1993):

- **Top matches schedule**: For a given set of rounds R, no top match can take place at any round $r \in R$.
- **Top match distance**: Two top matches cannot take place at distance smaller than a given value `TopMatchDistance`.
- **Top opponent distance**: Any team cannot match two top teams at distance smaller than a given value `TopOpponentDistance`.

The two parameters `TopMatchDistance` and `TopOpponentDistance` are given at global level; that is, they are the same for all teams. Their typical value can be 3 and 2, respectively.

We split teams in two groups: the top teams (usually 3 or 4) and the other ones. A finer grain grouping is also possible, and more complex constraint types can be considered. For example, Schreuder (1993) proposes (although he does not pursue this idea) to divide teams in three groups —strong, medium, and weak teams— and looks for schedules that alternate matches with teams belonging to the three groups.

3. Two-Step Approach

We propose a solution of the round robin tournament scheduling problem based on two steps: First, determine a *tournament pattern*, i.e. a complete tournament in which numbers from 1 to $2n$ are used as dummy teams. Second, associate all actual teams with distinct numbers in the pattern.

The total number of breaks is completely determined by the tournament pattern. Therefore, it is in the first step that we take care of minimizing such number. At the same stage, we also ensure that the tournament pattern is done so that there is a way to satisfy the complementarity constraints. All other constraints are not considered at this stage and they are dealt with in the second step.

In the second step, we take into account the actual constraints that involve the specific teams (ordinary and special ones), trying to satisfy the hard ones and minimize the total penalty of the soft ones.

3.1. Step 1: Determine a tournament pattern

The problem of determining the tournament pattern is related to the problem of finding an *1-factorization* of a complete (undirected) graph

Round 1	1-6	2-5	4-3
Round 2	6-2	3-1	5-4
Round 3	3-6	4-2	1-5
Round 4	6-4	5-3	2-1
Round 5	5-6	1-4	3-2

Figure 1. The canonical pattern for $2n = 6$

(Mendelsohn and Rosa, 1985). That is, given the complete graph K_{2n} we must partition it in a set of $2n - 1$ sets of n arcs (called *1-factors*), such that in each set the arcs are pairwise non adjacent.

Each arc represents a match and each 1-factor a round. Therefore, giving an order to the 1-factors and assigning home or away teams for each match, a 1-factorization can be turned into a tournament pattern.

de Werra (1981) proved that there cannot exist a tournament pattern for $2n$ teams with less than $2n - 2$ breaks, and he supplied a formula for constructing a pattern with exactly $2n - 2$ breaks, that he called the *canonical pattern*. In the canonical pattern, for each team t_1 there exists a unique team t_2 such that t_1 and t_2 have a *complementary schedule*. Pairs of teams with complementary schedules are called *complementary pairs*. The complementary pairs of the canonical schedule are $(1, 2n)$ and $(i, i + 1)$ for $i = 2, 4, \ldots, 2n - 2$.

The construction of the canonical pattern of any n is shown in Appendix A. The canonical schedule for $2n = 6$ is shown in Figure 1, where the order of the teams determines the location of the match: The first team plays home and the second one away. The complementary pairs are $(1, 6)$, $(2, 3)$, and $(4, 5)$.

Most of the national football tournaments involve a double round robin, such that the second round robin is a copy of the first one with home and away teams swapped. To create a schedule for the double round robin, the canonical pattern is not suitable because two teams (numbers $2n - 2$ and $2n - 1$) have two consecutive breaks (i.e. three consecutive matches home or away). Specifically, they occur in the last round of the first round robin and in the first round of the second one. In addition, the same two teams play the last two games in the same location, which is something sportively not fair.

For this reason, we consider the *modified canonical pattern* defined in (de Werra, 1981, Prop. 4), which is obtained from the canonical

Round 1	1-6	2-5	4-3
Round 2	6-2	3-1	5-4
Round 3	6-3	4-2	1-5
Round 4	4-6	5-3	2-1
Round 5	6-5	1-4	3-2
Round 6	6-1	5-2	3-4
Round 7	2-6	1-3	4-5
Round 8	3-6	2-4	5-1
Round 9	6-4	3-5	1-2
Round 10	5-6	4-1	2-3

Figure 2. The modified canonical pattern for $2n = 6$

one by reversing the orientation of the last three matches of the team number $2n$. Such pattern overcomes the above limitations, since it has no consecutive breaks and no breaks in the last round. In addition, it has exactly $6n - 6$ breaks for the whole double round robin, which is the minimum (de Werra, 1981, Prop. 3). Furthermore, it retains the property that teams have pairwise complementary schedules. The complementary pairs are $(1, 2n - 1)$, $(2n - 4, 2n)$, $(2n - 3, 2n - 2)$ and $(i, i + 1)$ for $i = 2, 4, \ldots, 2n - 6$. The full double tournament for $2n = 6$ is shown in Figure 2. Complementary pairs are $(1, 5)$, $(2, 6)$, and $(3, 4)$.

The modified canonical pattern is therefore suitable for the solution of our problem. Obviously, other patterns (having the required features) can also be used in place of the modified canonical one. In particular, we can think of patterns that satisfy some other requirements. For example, the patterns defined by Russell (1980) take care also of the so-called *carry-over effect*; that is, they avoid that a team plays too often with teams that played in the previous round with a specific team. Unfortunately, the patterns defined by Russell do not include home-away selection since they are meant for a tournament on a single site. Nevertheless, home and away teams can be assigned to them (in a way that minimizes the number of breaks) adapting the method proposed by Wallis (1983) to the double round robin case.

Therefore Russell's patterns are a possible alternative to the modified canonical pattern. In addition, many national football federations have their standard patterns which are used for all tournaments organized by the league. Therefore, they enforce the use of such patterns for the tournament.

In any case, it is worth remarking that the second step is completely independent of the choice of the specific pattern in use.

3.2. Step 2: Team Assignment

Given a fixed pattern, the second step of our approach aims at finding a matching between the actual teams and the numbers (dummy teams) appearing in the pattern.

The team assignment problem is a bipartite graph matching, which is a well-studied problem (see e.g., Hopcroft and Karp, 1973). However, we have to take into account our constraints, and the way they affect the structure of the problem.

Hard availability constraints force a given team not to be assigned to any number that plays in the undesired location at the given round. Therefore, constraints of this type simply remove some arcs from the complete bipartite graph.

Hard mating constraints require that a given pair of teams (t_1, t_2) is not assigned to any of the pairs of numbers that compose a given round r. Therefore, they can be reduced to a set of constraints, that we call *pair-inequality* constraints, stating that a given pair of teams (t_1, t_2) cannot be simultaneously assigned to a given pair of numbers (m_1, m_2).

Hard complementarity constraints require that a given pair of teams (t_1, t_2) is assigned to one of the complementary pairs of numbers. Such constraints can also be reduced to a set of pair-inequality constraints stating that (t_1, t_2) must be different from any pair but the complementary ones.

Hard fairness constraints require that a given pair of teams (t_1, t_2) is different from all the pairs that have more than the given number m of games together. Therefore, they also reduce to a set of pair-inequality constraints.

Triples constraints force triples of teams to be not simultaneously assigned to triples of numbers that are in the same location for at least one round. Since all such triples of numbers can be easily precomputed from the given pattern, triple constraints reduce to *triple-inequality* constraints which are the variant of pair-inequality with three teams.

All top teams constraints can be verified based on the assignment given to the top teams alone. Therefore, assuming that there are t

top teams (typically 3 or 4), all top teams constraints together can be reduced to a set of *tuple-inequality* constraints.

Regarding the soft constraints, all of them can be embedded in the objective function, which is the function that returns, for each feasible matching, the associated total penalty. In fact, all types of (soft) constraints can be easily computed when the complete matching is given.

Summing up, the problem we have to face in the second step is a *minimum-cost* matching problem in a (not necessarily complete) bipartite graph with tuple-inequality constraints.

It is easy to see that finding the optimal solution of the team assignment problem does not ensure to reach the optimal solution in the general case. Possible techniques to solve optimally the general case will be briefly discussed in Section 8.

4. Computational Complexity

As already mentioned, computing the solution of the tournament scheduling problem in the general case and in the two-step approach are two distinct problems. We now discuss the complexity of both problems.

4.1. Complexity of the Two-Step Approach

Regarding the complexity of the two-step approach, we can easily recognize that the modified canonical tournament pattern can be generated in polynomial time ($\mathcal{O}(n^2)$). Regarding the complexity of the team assignment problem, it is easy to see that, for a given matching, the objective function can be computed in polynomial time. Conversely, we prove that the underlying decision problem —"does a matching satisfying all the hard constraints exist?"— is NP-complete. To this aim, we have to prove that the problem is in NP and that it is NP-hard.

The NP membership is trivial, since every matching can be easily generated and verified in non-deterministic polynomial time. Regarding the NP-hardness, it is well known that bipartite graph matching is a polynomial problem (Hopcroft and Karp, 1973). Conversely, Itai *et al.* (1977) proved that the "restricted" bipartite graph matching is NP-complete, where restricted means that one can express constraints of the form: For a given set of arcs E at most r of them can be in the matching. Moreover, Itai *et al.* in their NP-completeness proof (which is a reduction from the SAT problem) make use *only* of constraints of a special type where E has cardinality 2 and $r = 1$. That is, they consider only a set of restrictions of the form: Between two arcs of

the graph, at most one can be part of the matching. Therefore, they implicitly proved that bipartite graph matching with this special type of restriction is also NP-complete. Our pair-inequality constraints are exactly restriction of the special type; in fact, the constraint that t_1, t_2 cannot be simultaneously assigned to m_1, m_2 is equivalent to state that at most one of the arcs (t_1, m_1) and (t_2, m_2) can be in the matching. Therefore, we can conclude that the team/number matching problem with pair-inequality constraints is NP-complete.

The fact that pair-inequality constraints make the problem NP-complete gives us an intuition of the hardness of the team assignment problem, but it does not imply that the team assignment problem is NP-complete. In fact, from the team assignment problem we cannot generate arbitrary pair-inequalities, but only unbreakable sets of them. The full proof of the NP-hardness of team assignment requires more complex machinery and it is provided in Appendix B.

4.2. Complexity of the General Problem

Now we discuss the complexity of the overall tournament scheduling problem. In particular, we consider the underlying decision problem —"does a tournament satisfying all the hard constraints exist?"— and we prove its NP-completeness.

We first prove that it is in NP. To this aim, we can think of a tournament as a table of quadratic size each entry of which is one of the teams. Such table can be guessed in polynomial time. The check that it is indeed a legal tournament amounts to verify that every team appears in every round and that every team plays with all other teams. It is easy to see that both these conditions, plus our hard constraints, can be verified in polynomial time, therefore the problem is not harder than NP.

Unfortunately though, there is no known way to enumerate all the possible tournament patterns in a computationally tractable way. On the graph-theoretic side, it is not even known which is the number of non-isomorphic 1-factorizations of the complete graph K_{2n} (independently of the orientation). To this respect there are some isolated results: It is known that for $2n = 2, 4, 6$ there is only one equivalence class of 1-factorizations. For $2n = 8$ there are 6 non-isomorphic 1-factorizations (Wallis *et al.*, 1972). Gelling and Odeh (1973) proved, by exhaustive computer construction, that for $2n = 10$ they are exactly 396. Lindner *et al.* (1976) found an exponential lower bound for such number, which proves that the number goes to infinity with n.

Rosa and Wallis (1982) introduce the notion of *prematurity*: Calling *round* any set of n matches that involve all teams, and calling *partial*

tournament any set of k rounds in which no match is repeated twice, then a partial schedule is said *premature* if it is not possible to create other $n-1-k$ rounds so as to generate a full tournament.

They proved the existence of premature schedules of $k > n$ rounds for all $n \geq 5$. They also proved that for $n \geq 4$ a partial tournament of 3 rounds is never premature, i.e. it can always be completed. Conversely, Colbourn (1983) proved that it is NP-complete to decide whether a partial tournament is not premature.

Based on Colbourn's result, we can infer that our tournament scheduling problem is NP-hard. In fact, *non prematurity* can be polynomially reduced to tournament scheduling with mating constraints.

For each round r of a partial tournament of k rounds, for each match t_i-t_j in round r we pose that constraint that team t_i cannot play at round r with all teams but t_j. Such constraints forces the tournament schedule to have the first k rounds exactly equal to the partial tournament, and therefore the problem results in scheduling the remaining rounds of the tournament (without further constraints). It easily follows that the partial tournament is premature if and only if the tournament scheduling problem has a solution. Thus, we can conclude that the decision problem for tournament scheduling is NP-complete.

5. Algorithm and Implementation

As already mentioned, our approach is to use a fixed tournament pattern and to solve the associated team assignment problem. To this aim, we use the modified canonical pattern mentioned in Section 2. For reasons that will be explained below, we rename the numbers appearing in the pattern in such a way that i and $i+1$ (for $i = 1, 3, \ldots, 2n-1$) have complementary schedule.

5.1. Constraint Logic Programming with Finite Domains

The program is implemented using the finite domain library of ECL^{i}PS^e. Finite domain variables are associated with a finite set of values (the domain) which represents all its possible instantiations. Variable domains can be seen as monadic predicates attached to variables; however they are dealt with at unification level instead of at resolution level as standard monadic predicates (see e.g., Van Hentenryck, 1989, Jaffar and Maher, 1994).

Finite domain constraints, like equality "`#=`", inequality "`##`", and disequality "`#<`", are processed based on the domain of the variables involved. They can succeed, fail, or *delay* depending on the current

state of the domain of the variables. Delayed goals are collected in the *constraint store* which affects the future dynamics of the variables involved.

For example, suppose that `A`, `B`, and `C` are three (uninstantiated) finite domain variables and their domains are the integer intervals `1..5`, `1..3`, and `5..7`, respectively. Then, the constraint `B #> C` would fail, whereas the constraint `B #< C` would succeed. Conversely, the constraint `A #< B` would delay, however in the mean time the domain of `A` is reduced to the interval `1..2` and the domain of `B` to the interval `2..3`. The reduction of the domain of `A` and `B` might affect the domain of other variables involved in delayed goals. In fact, any time the domain of one of the variables is reduced, the constraint is *woken* and the domain of the other variables is reduced consequently.

In this way, the constraint store can give a good pruning in the search space for variables to be instantiated for the solution of the problem.

5.2. General Program Architecture

The high level predicate definition of our program is the following:

```
sportSchedule(NbrTeams):-
        createDataStructures(NbrTeams),
        stateDomains(NbrTeams,TeamVars),
        stateConstraints(NbrTeams,TeamVars),
        generateValues(NbrTeams,TeamVars),
        printReport(NbrTeams,TeamVars).
```

In the first phase, by means of the invocation of the predicate `createDataStructures`, the program builds the patterns based on the number of teams and it declares and initializes all the auxiliary data structures associated to the pattern. The auxiliary data structures are used for a fast retrieval of all the information related to the pattern. For example, for each pair of numbers, we store the number of times the corresponding teams would be in the same location (home or away). Such structures are implemented using the ECL^{i}PS^e *array* facilities, which work much more efficiently than regular lists in standard logic programming languages.

In the second phase, through the predicate `stateDomains`, each team t is associated with a finite domain variable `T`, whose value corresponds to the number that the team gets in the tournament pattern. All variables are stored in a list, called `TeamVars`, whose length is the number of teams ($2n$), stored in the variable `NbrTeams`. Each variable of the list is associated with a domain, which is the integer interval from `1` to `NbrTeams`.

In the third phase, based on the hard constraints of the problem, we state, by means of the predicate `stateConstraints`, the constraints on the finite domain variables. This phase is explained in details in Section 5.3.

The fourth phase is the team assignment. This is the only phase in which backtracking takes place. The choice of the order for instantiating the variables is crucial for the efficiency of the algorithm. This phase is described in Section 5.4.

When the `generateValues` predicate has traversed the entire search space, the current best solution is passed to the predicate `printReport` which displays the full tournament, with the list of all soft constraints violated.

5.3. Constraint Definition

The fact that each team must be assigned to a different number, and thus that all values must be different from each other, is expressed by a call of the built-in `alldistinct(TeamVars)`, which generates inequality constraints between all pairs of constraints in the list `TeamVars`.

Availability constraints are taken into account simply by removing from the domain of a variable the numbers that in the pattern play in the location (home or away) where the team cannot be. For example, if team t cannot play home at round r, the program retrieves all the numbers that play at home at round r —which are stored at location r of the auxiliary array `HomeTeams`— and deletes all the corresponding values from the domain of the variable `T` (by means of the built-in `dvar_remove_element`).

As already stated, each mating constraint reduces to a set of pair-inequality constraints. Pair-inequality constraints are enforced by avoiding that the given pair of variables `T1,T2` are simultaneously instantiated with the given pair of values `V1,V2`. Exploiting the fact that the domain of the variables is bounded by the value `NbrTeams`, a pair-inequality constraint can be expressed using a single primitive inequality constraint in ECL^{i}PS^e in the following way:

```
T1 * NbrTeams + T2 ## V1 * NbrTeams + V2
```

The way constraints are dealt with in ECL^{i}PS^e ensures that if `T1` (resp. `T2`) is instantiated to `V1` (resp. `V2`), the value `V2` (resp. `V1`) is removed from the domain of `T2` (resp. `T1`). Conversely, if `T1` (or `T2`) is instantiated to a different value, the constraint is immediately satisfied (and thus discarded) independently of the value of `T2`. For example, if the number of teams is 10 and `V1` and `V2` are respectively 6 and 3, we state

the constraint `T1 * 10 + T2 ## 63`. If at a certain point of the computation, `T2` is instantiated to 3, the constraint is woken, instantiated to `T1 * 10 + 3 ## 63`, and simplified to `T1 ## 6`. Therefore, the value `6` is removed from the domain of `T1`. If `T2` is instantiated to `5`, the constraint reduces to `T1 * 10 ## 58` which is automatically satisfied and discarded.

Such approach gives much more pruning than just checking the violation of the constraint when both variables are instantiated, which can be achieved with conventional logic programming techniques.

Triple-inequality constraints are treated in an analogous way. Specifically, the constraint that (t_1, t_2, t_3) must be different from (v_1, v_2, v_3) is implemented by

```
T1 * NbrTeams * NbrTeams + T2 * NbrTeams + T3   ##
V1 * NbrTeams * NbrTeams + V2 * NbrTeams + V3
```

Complementary constraints in principle can also be reduced to pair-inequality constraints. However, for efficiency reasons they are treated differently. Nevertheless, they are also reduced to primitive finite domain constraints. In particular, since we renamed the pattern in such a way that values i and $i + 1$ (for $i = 1, 3, \ldots, 2n - 1$) have complementary schedules, the constraint that teams t_1 and t_2 must be complementary simple reduces to the following equality constraint

```
T2 #= T1 + 1
```

along with the constraint that `T1` has an odd value.

Notice that this way we have imposed that `T1` has the lower value and `T2` the higher. We can proceed this way (applying the general principle of eliminating symmetric cases whenever it is possible) only if there are no constraints that involve the single variables `T1` and `T2`.

Conversely, if there are other constraints on `T1` and `T2`, it is necessary to try also the dual assignment (i.e. `T1 #= T2 + 1`, with `T2` odd). In this case, we have a disjunctive constraint involving the variables `T1` and `T2`. Such constraints are dealt with by using the *generalized propagation library* Propia of ECL^{i}PS^e (ECRC, 1995a, Chapter 6), which implements a form of *constructive disjunction*.

Using constructive disjunction in Propia, choices due to disjunction are delayed as much as possible; however, before making the choice, the system extracts useful information common to the two branches. For example, suppose that `T1` and `T2` are finite domain variables with current domains respectively `3..5` and `1..10`, then a disjunction of the form

```
T1 #= T2 + 1 ;  T2 #= T1 + 1
```

is delayed until one of the two variables is "touched" (Le Provost and Wallace, 1993), but in the mean time the domain of `T2` is automatically reduced to `2..6`.

Special constraints are not considered at this stage for reasons that will be explained in the sequel.

5.4. Value Generation

Since the most constrained variables are those corresponding to the top teams, we start instantiating them. For the remaining teams, we split them in those on which some constraints (hard or soft) are stated and those that are completely unconstrained. For the latter ones, called *free teams*, any assignment is feasible and their assignment does not affect the objective function. For this reason, we assign the values for free teams after the regular ones so as to reduce the number of variables upon which backtracking is necessary.

The definition of the predicate `generateValues` is the following

```
generateValues(NbrTeams,TeamVars):-
   splitTeamVars(TeamVars,TopTeams,RegularTeams,FreeTeams),
   generateValuesForTopTeams(TeamVars,TopTeams)
   generateValuesForRegularTeams(TeamVars,RegularTeams),
   generateValuesForFreeTeams(FreeTeams).
```

The predicate `splitTeamVars` separate top team variables and free team variables from the variables of the rest of the teams, called *regular teams*.

For assigning top teams, we use the predicate `generateValuesForTopTeams` which makes use of a simple *generate and test* procedure. That is, an assignment for all top teams is generated before testing it against the special constraints.

This way of treating special constraints has been experimentally proven to be more effective, and it is intuitively justified by the fact that top teams are few and the number of feasible assignments is also small, whereas the cost of checking the special constraints can be relatively high. However, it is worth noticing that the ordinary constraints in the store are automatically taken into account and they prevent the search space for the top teams to become too large. For example, they ensure that top teams are assigned to different numbers and they satisfy the availability constraints.

For each feasible assignment for the top teams, we look for an assignment for the regular teams with the predicate `generateValuesForRegularTeams`. This is the computationally hard part of the program, and it is dealt with a branch and bound algorithm. Therefore, in this phase,

pruning takes place not only based on constraints accumulated in the store by the `stateConstraints` predicate, but also due to the binding activity for the branch and bound scheme. That is, a backtracking can occur either because the domain of a variable becomes empty or because of the value of the objective function based on the current best solution.

Variables are chosen one at a time to be instantiated to a value belonging to its domain. For the selection of the next variable to be instantiated, we use the `deleteffc` built-in, that retrieves the variable with the smallest domain and (in case of equal size) the most constrained one.

The choice of the possible value for the selected variable is done by computing a lower bound of the objective function for each possible partial solution. In details, for each soft constraint the evaluation returns its penalty if the constraint is violated and 0 if it is not violated. For the constraints that cannot be checked because the variables involved are not instantiated yet, we compute a lower bound of their penalty based on the current domain of the variables. Obviously, the evaluation takes into account also the variables that are automatically instantiated due to the constraints and not only those instantiated by the labeling process. Based on such evaluation, values are sorted in ascending order, to be selected one at a time upon backtracking. After each instantiation, if the value of the objective function for the given (partial) solution is higher than the current best (if any), then the evaluation fails and the program backtracks.

When a solution has been found for top teams and regular teams, the predicate `generateValuesForFreeTeams` generates values for the free teams (without backtracking) using the `labeling` built-in predicate, which chooses variables in the order they appear in the list, and instantiates it with the minimum of its current domain.

6. Experimental Results

We now present the experimental results for the team assignment problem. In this section, when we write "optimal" solution we refer to the solution of the team assignment problem and not to the general tournament scheduling problem.

6.1. General Results

We experiment with instances of size $2n = 12$, 18, and 20, which correspond to the size on real cases. The number of top teams is fixed to 2,

Table I. General results for the team assignment problem

teams	Normal		Hardened		Relaxed	
	time	# btr	time	# btr	time	#btr
12	4.2 secs	16,940	1.6 secs	3,965	12.6 secs	49,787
18	13 min	3,018,321	6.1 secs	13,545	115 min	24,103,431
20	26 min	3,675,750	10.1 secs	20,113	140 min	22,950,953
30	7 hours	$> 10^8$	23.2 secs	57,868	28 hours	$> 10^8$
40	8 hours	$> 10^8$	35.3 secs	91,276	$>$ 48 hours	$> 10^8$

3, and 4, respectively; no free teams are used. We added the cases for $2n = 30$ (5 top teams, 2 free teams) and 40 (6 top teams, 6 free teams) to see how the program scales up. For each size, we run the program on instances of three different types. The first type is a regular instance created along the line of real data. The second type is obtained "hardening" the soft constraints that are satisfied by the optimal solution of the corresponding instance of the first type. The third type, on the contrary, is an instance with no hard constraints (except the special ones), obtained from the first type by relaxing most of the hard constraints into soft ones.

Table I summarizes the results, which are obtained running the program on a Sun `SPARCstation 4`, and show the running time and the total number of backtracks. The results show that, as expected, hard constraints give much more pruning than soft ones. In fact, the *a-priori* pruning given by domain reduction is more effective than the pruning given by the failure due to the bounding capabilities of the branch and bound. Therefore, in order to improve the efficiency of the program, it is advisable to include as many hard constraints as possible. For example, if for two given teams they both wish to have a complementary schedule it is reasonable to assign it to them as a demand, even though they do not share the same stadium.

They also show that for regular size (≤ 20) the program runs reasonably fast, whereas for larger instances it works fast only when many hard constraints limit the search space. Therefore, if such large instances occur, a more efficient —possibly approximate— treatment of soft constraints would be necessary.

As shown in Table I, for 18 teams using real data,[1] it takes about 13 minutes to generate the optimal solution. Additional runs on different

[1] Kindly supplied by Jan Schreuder for the Dutch "Top League" for years 1994-95 and 1995-96.

data obtained making some perturbation show that such value varies between 4 minutes and 30 minutes.

The only optimal solution method available is the diagnostic system described in (Bakker *et al.*, 1993), which solves instances of the same size in about 25 hours of cpu time. The method proposed by Schreuder (1992) instead takes about 2 minutes of cpu time (plus some manual adjustments). However, it must be clear that Schreuder uses an incomplete procedure which gives no guarantees about the optimality of the solution. Furthermore, Schreuder provides no approximation result about the quality of the solutions found by his algorithm. Our method instead provides the optimal solution even for general tournament scheduling problem at least for the cases in which the pattern is given in advance. In addition, experiments done with a limited number of patterns show that the optimal solutions found with different patterns are very close to each other in terms of the value of the objective function (within 10%).

A solution method which is not based on the two-step approach, but rather builds the pattern from scratch, has been proposed by McAloon *et al.* (1997). As reported by the authors, such an approach turned out to be feasible only for at most 14 teams.

6.2. Impact of the Features of the Algorithm

The critical issues of our program (and of constraint logic programming in general) are the ordering of the variables and the selection of the appropriate value, within the current domain of the variable, for the instantiation.

Table II shows the impact of such issues in the performances of the program. In particular, it shows, on instances of 18 teams, the performances (in seconds) of the combination of three different variable selection strategies with three different value ordering.

The variable selection strategies considered are: (*i*) choose the variable with the smallest current domain, breaking ties by choosing the one with the largest number of constraints attached to it (built-in `deleteffc`), (*ii*) choose the variable with the smallest current domain, breaking ties arbitrarily (built-in `deleteff`), (*iii*) choose variables in a fixed static order (built-in `labeling`).

The value ordering strategies are: (*i*) order values based on an estimation of the objective function, (*ii*) sort values (in an increasing ordering), (*iii*) order values randomly.

The use of the built-in `deleteffc` for selecting variables in the second phase gives a huge speed-up with respect to the naive `labeling`

Table II. Performances of variable selection and value ordering strategies

Variable Ordering / Value Selection	best first	smallest first	random
Smallest domain most constrained	775.7	879.2	951.0
Smallest domain	1194.0	1464.9	1378.3
Static order	11870.2	13517.0	13254.3

```
Algorithm TournamentScheduling
Input Instance : TournamentSchedulingInstance;
Output Solution : AssignmentSolution;
begin
   Solution := SolveApproximate(Instance);
   while not Satisfying(Solution)
   begin
      Instance := ManuallyAdjustSpecification(Instance);
      Solution := FastReviseSolution(Instance,Solution)
   end
   Solution := SolveExactly(Instance);
end.
```

Figure 3. The interactive algorithm

built-in predicate, which chooses variables in the order they appear in the list.

Regarding the value ordering issue, our selection based on the objective function also gives better performances than both the sorting choice (built-in `indomain`), and a random ordering of the values.

7. Interactive System

The ability to work interactively is widely recognized as crucial for scheduling systems. For our problem, although each instance can be solved optimally in reasonable time, in order to solve a real case, the run must be repeated several times so as to get sensibility on constraints and penalties. Therefore, it is necessary to have a fast (possibly incomplete) method that runs in a few seconds, that allows the user to play interactively with the constraints and the corresponding solutions. Specifically, the typical session with the system has the structure shown in Figure 3.

7.1. Fast Sub-Optimal Construction

The function SolveApproximate is meant to give a sub-optimal assignment in short time (say in 2-3 minutes). One easy way to solve this problem is to stop the search when time is expired and to return the current best solution. A different way is to reduce the branching factor during the branch and bound search (see Ginsberg and Harvey, 1992). That is, we might not consider all possible values for the selected variable, but only the best k ones, where k is a selected parameter. In (Ginsberg and Harvey, 1992), k is iteratively increased so as to retain completeness of the procedure. Conversely, in order to have a fast (incomplete) procedure, k must be selected based on a compromise between efficiency and completeness.

Our experimental results show that the value $k = 3$ almost never misses the optimal solution, and gives a speed-up of 2 (i.e. it halves the computational time). The value $k = 2$ gives a speed-up of approximately 5, but in a few cases does not find the optimal solution. We therefore use a branching factor of 2 in order to implement the procedure SolveApproximate.

7.2. Solution Revision

In order to implement the function FastReviseSolution we make use of a *local search*. Local search techniques are a family of general-purpose techniques for the solution of optimization problems. They are based on the notion of *neighbor*. Consider an optimization problem, and let S be its search space and f its objective function to minimize. A function N, which depends on the structure of the specific problem, assigns to each feasible solution $s \in S$ its *neighborhood* $N(s) \subseteq S$. Each solution $s' \in N(s)$ is called a neighbor of s.

A local search technique, starting from an initial solution s_0, enters in a loop that *navigates* the search space, stepping iteratively from one solution to one of its neighbors. We call *move* the modification that transforms a solution to one of its neighbors.

In our case, the initial solution s_0 is the solution of the problem considered in the previous iteration, and a local move consists in swapping the assignments given to two different teams.

Local search techniques are especially suitable for our purpose, since they allow to revise the given solution, based on the new constraints, without recomputing it from scratch.

Specifically, we implemented a hill climbing procedure based on the *Min-Conflict Hill Climbing* (MCHC) technique defined by Minton *et al.* (1992). That is, a move consists in randomly selecting a team t, and

swapping the assignment for t with the assignment of another team s, choosing s in such a way to minimize the number of infeasibilities and —with less priority— the objective function.

MCHC allows also for *sideways moves*, i.e. moves that leave the value of the objective function unaltered. Therefore this method has the feature of being able to follow descending paths that pass through *plateaux*. That is, if the search lands in a plateau, it is able to move within it, and might get down from it through a solution different from the one from which it reached the plateau.

Accepting sideways move, the algorithm can run for infinite time, we therefore fix a maximum number of iterations so as to keep its running time within a reasonable amount of time (about 1 minute).

Although MCHC has the capability of navigating plateaux, it is inevitably trapped by strict local minima. More sophisticated local search techniques (like tabu search and simulated annealing) also accept worsening moves and allow one to escape from strict local minima. We do not discuss their use in this paper, however we believe that, due to the limited time granted to the algorithm, more complex ones would not give any improvement. This conjecture is supported by preliminary experimental results with tabu search.

8. Conclusions and Future Work

We have presented a constraint-based branch and bound algorithm for a sport scheduling problem. Our procedure uses exponential time in the worst-case. However, since the problem is NP-complete, such complexity is unavoidable.

We have also discussed a local search procedure that complements the branch and bound algorithm, allowing the resulting system to be a useful tool for interactive runs.

Despite its theoretical complexity and despite the common opinion that this kind of problems cannot be solved in an exact way (see e.g., Schreuder, 1993), the problem turned out to be relatively easy to handle using constraint programming. In fact, the solution program is considerably short and quite straightforward to write. Moreover, it is flexible, readable and easy to maintain.

We do not claim that all tournament scheduling problems can be easily solved using constraint logic programming. There are some problems that involve more than one league (see e.g., de Werra *et al.*, 1990, Nemhauser and Trick, 1997), and others that are based on the minimization of traveling costs for the teams (see e.g., Campbell and San Chen, 1976). Such more complex sport scheduling problems general-

ly require specialized optimization techniques (see e.g., Costa, 1995, Ferland and Fleurent, 1991).

As already mentioned, the two-step approach does not ensure to find the optimal solution for the general problem. Theoretically, there are two possible approaches to the general problem.

The first approach would be to construct directly a complete tournament respecting the above constraints and ensuring a minimum number of breaks. In that case, the number of breaks can be either a soft or a hard constraint. This approach, however, seems to be extremely expensive from the computational point of view, and thus absolutely intractable for practical cases.

A more promising idea would be a generalized two-step approach, based on the enumeration of all possible patterns. However, as mentioned in Section 4, this approach seems to be extremely hard to formalize and solve, especially due to the lack of suitable graph-theoretic results. In particular, the hardness of this approach is testified by the fact that it is not even known how many different patterns exist for 12 or more teams.

An intermediate solution, which we plan to implement in the future, is to collect a number of different patterns and to look for the global minimum using one of them. The main issue of this approach is to identify those patterns that are different enough to each other with respect to their ability to satisfy our type of constraints.

We also plan to work for improving further the efficiency of the program. To this aim, we plan to make use of the filtering algorithm proposed by (Régin, 1994), which allows the solver to provide a pruning associated to the `alldistinct` constraint more effective than the ECL^{i}PS^e built-in implementation. In addition, we want to look for a better upper-bound to the cost of a partial solution so as to give a larger pruning in the branch and bound procedure based on the soft constraints.

Acknowledgements

I wish to thank Jan Schreuder and Krzysztof Apt for many fruitful discussions that contributed to the paper, Bruno Errico and Maurizio Lenzerini for useful comments on an earlier draft of the paper. All four anonymous referees provided me with very useful suggestions.

This work has been partly carried out while the author was visiting CWI in Amsterdam and it is part of the ERCIM fellowship Programme and financed by the Commission of the European Communities.

Appendix

A. Construction of the Canonical Pattern

We illustrate the formula for constructing the canonical tournament pattern for $2n$ teams defined in (de Werra, 1981). Each round i (with $i = 1, \ldots, 2n-1$) of the pattern is composed by the following matches:

- one match:

  $[2n,i]$ if i is even
  $[i,2n]$ if i is odd

- $\lfloor n/2 \rfloor$ matches:

  $[i+k,i-k]$ for $k = 1, 3, \ldots, 2 \cdot \lfloor n/2 \rfloor - 1$

- $\lfloor (n-1)/2 \rfloor$ matches:

  $[i-k,i+k]$ for $k = 2, 4, \ldots, 2 \cdot \lfloor (n-1)/2 \rfloor$

Each pair $[a,b]$ denotes a match in which a is the home team and b the away team. The number $i+k$ in the formula must be interpreted "modulo 2n-1" in the following way: if $i+k < 2n$ then it corresponds to $i+k$ itself, otherwise it corresponds to $i+k-(2n-1)$. Similarly, if $i-k > 0$ it corresponds to itself, otherwise it corresponds to $i-k+(2n-1)$.

B. NP-completeness of the Team Assignment Problem

We prove that the team assignment problem is NP-complete in the general case, in which the pattern is part of the input. This means that there exist a pattern that makes the team assignment problem NP-complete. We do not prove that the problem is NP-complete for any specific given pattern.

The proof of the theorem is a reduction from the propositional satisfiability problem to the team assignment problem, and it goes along the line of the proof of NP-completeness of the restricted bipartite graph matching given in (Itai *et al.*, 1977).

Before proving it we need the following construction that will be used in the proof.

Construction of the Pattern. Given two positive even numbers g and m we can construct a pattern for $g \cdot m$ teams in the following way.

First, construct a canonical pattern for g teams, which we call *group-pattern*. The numbers in the group-pattern are called *group-teams*. Each group-team i corresponds to m teams $i_1, i_2, \ldots, i_m$ in the full pattern.

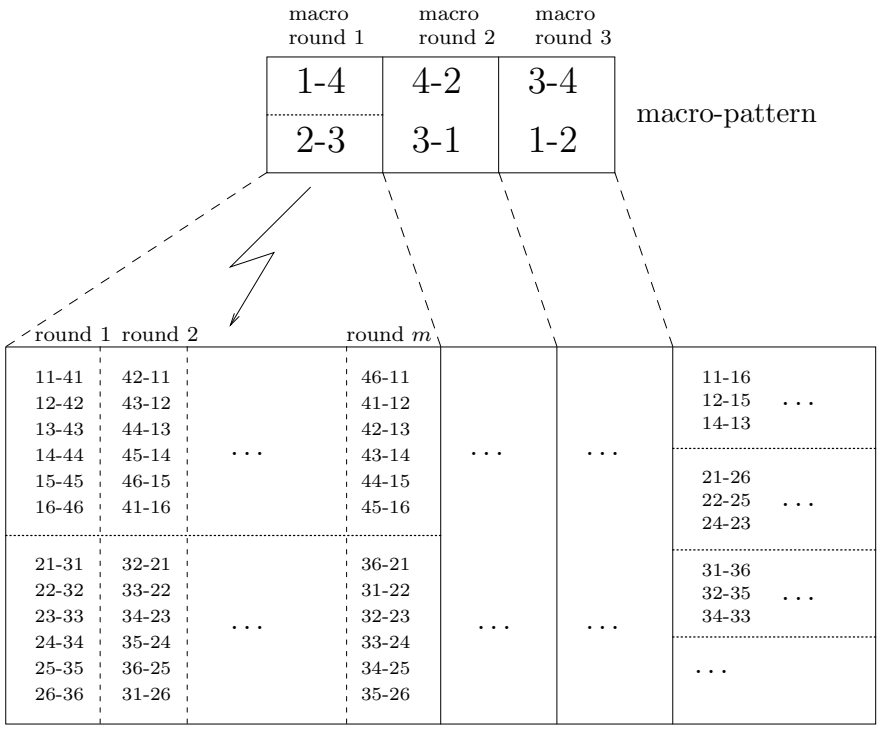

Figure 4. A pattern construction for $g = 4$ and $m = 6$

Second, from the group pattern, we generate the full pattern as follows. Each match i-j in the group-pattern is expanded to become a set of m rounds in which all teams $i_1, i_2, \ldots, i_m$ match all teams $j_1, j_2, \ldots, j_m$. A set of $m-1$ rounds are added at the end of the pattern in which teams in each group match among themselves.

The home and away teams are assigned arbitrarily, except for some of them as explained in the next step.

The construction is shown in Figure 4 for $g = 4$ and $m = 6$ (where team i_k is denoted by ik). □

Construction of the Reduction. Let $F = C_1 \wedge C_2 \wedge \cdots \wedge C_n$ be a CNF propositional formula, and $l_1, l_2, \ldots, l_q$ be all the letters appearing in F. Without loss of generality assume n to be an even number.

We create a pattern with $n \cdot 2q$ teams using the above construction with $g = n$ and $m = 2q$. Each clause C_i corresponds to group-team i. For each of the $n \times q$ pairs clause/letter (C_i, l_j), we consider two symbols d_{ij} and $\overline{d}_{ij}$, which correspond to the (dummy) teams in the pattern.

The $n \cdot 2q$ real teams to be assigned to the dummy ones described above are created in the following way:

– for each clause C_i $(i = 1, \ldots, n)$ in F we associate a real team t_i in the assignment problem;

– we add the remaining $n \cdot 2q - n$ teams, on which no constraints are associated.

We now define the constraints in the team assignment associated with F. Intuitively, there are posed in such a way that a team t_i, associated with a clause C_i, can be assigned only to the literals belonging to C_i; so that the assignment of t_i to d_{ik} (or $\overline{d}_{ik}$) corresponds to the truth assignment of l_k to True (or False), i.e., l_k (or $\overline{l}_k$) is the literal assigned to True which makes the evaluation of C_i equal to True.

For example, if $C_i = l_3 \wedge l_5 \wedge \overline{l}_8$, then the constraints are such that t_i can be assigned only to d_{i3}, d_{i5}, and $\overline{d}_{i8}$.

For each group-match t_i-t_j, we arrange the dummy teams $d_{i1}, \overline{d}_{i1}, \ldots, d_{iq}, \overline{d}_{iq}$ and $d_{j1}, \overline{d}_{j1}, \ldots, d_{jq}, \overline{d}_{jq}$ so that all the matches d_{ik}-$\overline{d}_{jk}$ and $\overline{d}_{ik}$-d_{jk} (for $k = 1, \ldots, q$) take place simultaneously at the first round, among those belonging to the expansion of the group-match i-j. Then we impose for each pair i, j the mating constraint that teams t_i and t_j cannot play at the first round of the set of rounds resulting from expanding the group-match t_i-t_j.

The above constraints ensure that if a team t_i is assigned to a dummy team d_{ik}, no team t_j can be assigned to the dummy team $\overline{d}_{jk}$. Therefore, if l_k is the literal whose assignment to True verifies the clause C_i, then the literal $\overline{l}_k$ cannot be the one that verifies clause C_j.

In order to state that a team t_i can be assigned only to the dummy teams corresponding to the literals in the clause C_i, we make use of availability constraints in the following way.

First, we ensure that a team t_i is not assigned to a dummy team t_{jk} or $\overline{t}_{jk}$ with $j \neq i$. Consider again the first round of the set of rounds corresponding to the group match i-j; in such round, all dummy teams d_{ik} and $\overline{d}_{ik}$ play in the same location, and the dummy teams d_{jk} and $\overline{d}_{jk}$ play in the opposite one. We impose that team t_i plays in the location of dummy teams d_{ik} and $\overline{d}_{ik}$, thus ruling out the assignment to dummy teams related to clause C_j. This is repeated for each pair i, j.

In order to ensure that a team t_i is assigned only to the dummy teams belonging to the clause C_i, for each clause C_i we choose one round r and we rearrange the location of the matches d_{ik}-d_{jh} so that only those k so that l_k belongs to C_i play home (similarly for the negated literal). Posing the constraint that t_i must play home in round r, we obtain the required limitation. Round r can be any round except for the first of each group-match (whose home-away locations have already been fixed), and the last $2q - 1$ ones (which correspond to

matches between teams belonging to a single group-team and cannot be arranged freely). □

Correctness of the Reduction. Given a truth assignment that satisfies F, we can generate a feasible team assignment in the following way: for each clause C_i, select a literal l_k (or $\overline{l_k}$) that is assigned to True (at least one must exist). The team t_i is assigned to the dummy team d_{ik} (or $\overline{d}_{ik}$). The remaining teams are assigned arbitrarily to the remaining dummy teams. Due to the above construction, this assignment satisfies all constraints.

Given a team assignment that satisfies all the constraints, for each clause if team t_i is assigned to d_{ik} then l_k is assigned to True. If team t_i is assigned to $\overline{d}_{ih}$ then l_h is assigned to False. Due to the constraints posed in the above reduction, each letter is assigned only one value and each clause has at least one literal satisfied. Hence F is satisfied. □

References

R. R. Bakker, F. Dikker, F. Tempelman, and P. M. Wognum. Diagnosing and solving over-determined constraints satisfaction problems. In *Proc. of the 13th Int. Joint Conf. on Artificial Intelligence (IJCAI-93)*, pages 276–281. Morgan Kaufmann, 1993.

William O. Cain, Jr. The computer-assisted heuristic approach used to schedule the major league baseball clubs. In S. P. Ladany and R. E. Machol, editors, *Optimal Strategies in Sports*, pages 32–41. North-Holland, Amsterdam, 1977.

Robert Thomas Campbell and Der San Chen. A minimum distance basketball scheduling problem. In R. E. Machol, S. P. Ladany, and D. G. Morrison, editors, *Management Science in Sports*, pages 15–25. North-Holland, Amsterdam, 1976.

Charles J. Colbourn. Embedding partial Steiner triple systems is NP-complete. *Journal of Combinatorial Theory*, Series A 35:100–105, 1983.

D. Costa. An evolutionary tabu search algorithm and the NHL scheduling problem. *INFOR*, 33(3):161–178, 1995.

D. de Werra, L. Jacot-Descombes, and P. Masson. A constrained sports scheduling problem. *Discrete Applied Mathematics*, 26:41–49, 1990.

D. de Werra. Geography, games and graphs. *Discrete Applied Mathematics*, 2:327–337, 1980.

D. de Werra. Scheduling in sports. In P. Hansen, editor, *Studies on Graphs and Discrete Programming*, pages 381–395. North Holland, 1981.

D. de Werra. On the multiplication of divisions: The use of graphs for sports scheduling. *Networks*, 15:125–136, 1985.

ECRC, Germany. *ECL^{i}PS^e Extensions User Manual (Version 3.5.2)*, December 1995.

ECRC, Germany. *ECL^{i}PS^e User Manual (Version 3.5.2)*, December 1995.

J. A. Ferland and C. Fleurent. Computer aided scheduling for a sport league. *INFOR*, 29:14–25, 1991.

Eric N. Gelling and Robert E. Odeh. On 1-factorizations of the complete graph and the relationship to round robin schedules. In *Third Manitoba Conference on Numerical Math.*, pages 214–221, 1973.

M. L. Ginsberg and W. D. Harvey. Iterative broadening. *Artificial Intelligence*, 55(2-3):367–383, 1992.

J. E. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM Journal of Computation*, 2:225–231, 1973.

Alon Itai, Michael Rodeh, and Steven L. Tanimoto. Some matching problems for bipartite graphs. Technical Report TR93, IBM Israel Scientific Center, Haifa, Israel, 1977.

Joxan Jaffar and Michael Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

Thierry Le Provost and Mark Wallace. Generalized constraint propagation over the CLP scheme. *Journal of Logic Programming*, 16:319–359, 1993.

Charles C. Lindner, Eric Mendelsohn, and Alexander Rosa. On the number of 1-factorizations of the complete graph. *Journal of Combinatorial Theory*, Series B 20:265–282, 1976.

Ken McAloon, Carol Tretkoff, and Gerhard Wetzel. Sport league scheduling. In *Annual ILOG Optimization Users Conference*, 1997.

Eric Mendelsohn and Alexander Rosa. One-factorizations of the complete graph – a survey. *Journal of Graph Theory*, 9:43–65, 1985.

Steven Minton, Mark D. Johnston, Andrew B. Philips, and Philip Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.

George Nemhauser and Michael Trick. Scheduling a major college basketball conference. In *Proc. of the 2nd Int. Conf. on the Practice and Theory of Automated Timetabling*, pages 334–336, 1997.

Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *Proc. of the 12th Nat. Conf. on Artificial Intelligence (AAAI-94)*, pages 362–367, 1994.

Alexander Rosa and Walter D. Wallis. Premature sets of 1-factors or how not to schedule round robin tournaments. *Discrete Applied Mathematics*, 4:291–297, 1982.

K. G. Russell. Balancing carry-over effects in round robin tournaments. *Biometrika*, 67(1):127–131, 1980.

J. A. M. Schreuder. Constructing timetables for sport competitions. *Mathematical Programming Study*, 13:58–67, 1980.

J. A. M. Schreuder. Combinatorial aspects of construction of competition dutch professional football leagues. *Discrete Applied Mathematics*, 35:301–312, 1992.

J. A. M. Schreuder. *Construction of fixture lists for professional football leagues.* PhD thesis, Department of Management Science, The University of Strathclyde, Glasgow, December 1993.

T. H. Straley. Scheduling designs for a league tournament. *Ars Combinatorica*, 15:193–200, 1983.

Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

W. D. Wallis, A. P. Street, and J. S. Wallis. *Combinatorics: Room Squares, Sum-Free Sets, Hadamard Matrices.* Number 292 in Lecture Notes in Mathematics. Springer-Verlag, New York, 1972.

W. D. Wallis. A tournament problem. *Journal of the Australian Mathematical Society*, Series B 24:289–291, 1983.