Pseudo-Boolean optimisation for RobinX sports timetabling

Martin Mariusz Lester

Accepted: 3 May 2022 / Published online: 18 June 2022
© The Author(s) 2022

Abstract

We report on the development of Reprobate, a tool for solving sports timetabling problems in a subset of the RobinX format. Our tool is based around a monolithic translation of a sports timetabling instance into a pseudo-Boolean (PB) optimisation problem; this instance can be solved using existing pseudo-Boolean solvers. Once the tool has found a feasible solution, it can improve it using a second encoding that alters only the home/away pattern of games. We entered our tool into the International Timetabling Competition 2021. While it was effective on many instances, it struggled to cope with schedules involving large break constraints. However, among instances for which it could initially find a feasible solution, the combination of use of a portfolio of solvers, a range of variations on the encoding and the aforementioned local improvement process yielded an average reduction in solution cost of 23%.

Keywords Pseudo-Boolean constraints · Sports timetabling · International Timetabling Competition · RobinX

1 Introduction

Whereas previous instances of the International Timetabling Competition (ITC) have been based mainly around educational timetabling, the ITC 2021 was based around sports timetabling. The task for the competition was to produce double round-robin (2RR) tournament timetables for 16, 18 or 20 teams satisfying a mixture of soft and hard constraints. The goal was to produce a solution satisfying all hard constraints while minimising the sum of costs of violated soft constraints. Constraints were specified in a restricted form of the RobinX format; see the description of RobinX (Van Bulck et al. 2020b), the competition problem specification (Van Bulck et al. 2020a) and the competition report (Van Bulck et al. 2021) for full details.

We approached the problem using an encoding with Pseudo-Boolean (PB) constraints, which extend the ubiquitous family of Boolean satisfiability (SAT) constraints. In our previous experience of generating tournament schedules for mahjong tournaments (Lester 2021), we found that this was an effective way of generating a schedule satisfying a variety of complex and combinatorially hard constraints. The use of an existing constraint family, with existing solvers, removes the need to create a dedicated algorithm for solving the constraints and allows new constraints to be added easily. However, some care is still needed, as solvers can be sensitive to exactly how constraints are encoded. As PB is less well-known than SAT, Sect. 2.1 gives an overview of PB constraints. Section 2.2 discusses some relevant previous work on sports timetabling, including SAT-based approaches.

Our timetabling tool, Reprobate, uses a monolithic encoding of a RobinX instance into a PB instance, specifically a Weighted Boolean Optimisation (WBO) instance, as described in Sects. 3.1–3.3. It solves this using a portfolio of existing PB solvers, namely clasp (Gebser et al. 2012) and Sat4J (Berre and Parrain 2010), with a range of different settings. If a feasible solution is found, Reprobate extracts an initial timetable from this.

While the monolithic encoding finds many feasible solutions, they are far from optimal. To improve upon this, we use an approach from our previous work (Lester 2021). Reprobate improves the initial timetable by generating a second WBO instance in which the pairings of teams in each time slot is fixed, but their home/away pattern is not; Sect. 3.4 describes this encoding. Again, it solves this using a PB solver and extracts an improved timetable from the solution. At a high level, our approach has some similarities with the first-schedule-then-break method (Trick 2000).
Our tool is written as a series of Perl scripts that call existing solvers. We give some details of our choice of solver portfolio in Sect. 4.

We evaluate Reprobrate computationally on instances from the ITC 2021 in Sect. 5, looking particularly at the effect of using a portfolio and of some variations in the encoding in Sect. 5.3, and at the local improvement process in Sect. 5.5. In the context of the ITC 2021, its main weakness was handling large break constraints, which restrict the number of times teams may play consecutive games at home or consecutive games away. This is a known limitation of SAT-based approaches, for which we have implemented some mitigations from previous work (Horbach et al. 2012).

Reprobate is the first PB-based tool for solving sports timetabling problems presented in the RobinX format and, to the best of our knowledge, the first general-purpose sports timetabling tool that uses the OPB (Optimising with Pseudo-Boolean) file format and its associated solvers for Pseudo-Boolean Satisfaction (PBS), Pseudo-Boolean Optimisation (PBO) and Weighted Boolean Optimisation (WBO) problems. Both the tool (Lester 2021) and our preferred solver are available online under the open source MIT License, making it easy for others to use or to develop further.

2 Previous work

2.1 Pseudo-Boolean constraints

The pseudo-Boolean constraint satisfaction problem (PBS) (Manquinho and Roussel 2006) is a generalisation of the well-known Boolean constraint satisfaction problem (SAT). For Boolean variables $X_i$, a SAT constraint is the logical OR of literals $X_i$ or $\neg X_i$. Finding the solution to a set of SAT constraints is the canonical NP-complete problem. While there is no known polynomial-time solution method for SAT, there are practical, highly optimised solvers that can handle industrially relevant problems with millions of constraints. These solvers can sometimes be used as a “black box” that solves problems with little or no configuration. However, their performance is often highly dependent on exactly how a problem is encoded. Most leading SAT solvers are based around conflict-driven clause learning (CDCL), a search algorithm that identifies the cause of conflicting constraints, backtracks, then continues in a way that avoids the conflict.

A pseudo-Boolean (PB) constraint over Boolean variables (interpreted as integers 0 or 1) has the normalised form $\sum_i c_i \cdot X_i \geq w$, with integer coefficients $c_i$ and a positive integer degree $w$. By interpreting $X_i$ as $1 \cdot X_i$ and $\neg X_i$ as $1 + \neg 1 \cdot X_i$, it is easy to translate a SAT constraint into a PB constraint, rearranging the inequality to a normal form if necessary.

As a formalism for modelling real-world problems and discrete puzzles, where one often encodes whether an element is a member of a set using a Boolean variable, the advantage of PB constraints (compared with SAT) is that one can easily express constraints on the size of sets. For example, it is easy to express that $|A| \leq |B|$ or that $|A| \geq k$ or that $|A| = 1$. These kinds of constraints are extremely common when one wishes to express that an object occupies a particular position, or that all available positions are filled. In the case of sports timetabling, we may wish to express that a team must only play one game at a time, or that a team plays in every time slot.

The pure PBS problem can be extended in two main ways. Firstly, one can allow nonlinear constraints, such as $X_1X_2 + X_3X_4 = 1$. Here, $X_1X_2$ is true just if both $X_1$ and $X_2$ are true, and similarly for $X_3X_4$. Secondly, one can consider optimisation problems. In the simple Pseudo-Boolean Optimisation (PBO) case, a problem instance contains an objective function, such as $5X_1 + 2X_2$, which must be minimised. In the more general Weighted Boolean Optimisation (WBO) case, individual constraints can be assigned a cost, and the goal is to minimise the sum of costs of all violated constraints. For example, the hard constraint $X_1 \geq 1$ can be turned into a soft constraint with cost 5, which is denoted $[5] X_1 \geq 1$.

From 2005 to 2016, the Pseudo-Boolean Competition (Manquinho and Roussel 2006), a satellite to the high-profile SAT Competition, evaluated and ranked PB solvers, and helped to standardise an input format. PB solver technology has continued to advance since then, but the MaxSAT competition (Bacchus et al. 2019) has been more prominent. MaxSAT adds costs to SAT constraints, in a similar way to how WBO adds costs to PB. However, it does not allow for easy expression of cardinality constraints.

In the absence of nonlinear constraints, PB constraints are an equivalent formalism to 0–1 integer linear programming (ILP). However, there is a practical distinction between PB and 0–1 ILP in terms of the techniques used by the solvers and hence effective encoding techniques. PB is viewed as a generalisation of SAT and indeed, many PB solvers work by translating the constraints into SAT (for example by encoding an adder circuit) and using a CDCL SAT solver (Eén and Sörensson 2006). The library PBLib even provides tools for translating PBS and PBO problems into SAT and MaxSAT, respectively (Philipp and Steinke 2015). Conversely, 0–1 ILP is viewed as a restriction of mixed integer programming (MIP), and many 0–1 ILP solvers use techniques from linear programming (LP). Both kinds of solver may employ cutting planes reasoning.

The strengths and weaknesses of these solving techniques have been compared on several occasions. Coming from the MIP world, Berthold et al. (2008) develop the MIP solver SCIP, incorporating CDCL-style conflict analysis, and eval-
Mathematical puzzles. Rasmussen and Trick (2008) give an extensive survey of different problems in round-robin tournament timetabling and approaches to solving them.

2.2 s

There have been many different approaches to generating sports tournament timetables by computer. Van Bulck et al. (2020b) note that research publications tend to consist mainly of case studies focusing on one specific application, which makes them difficult to compare. Another common kind of publication focuses on abstract problems that are essentially mathematical puzzles. Rasmussen and Trick (2008) give an extensive survey of different problems in round-robin tournament timetabling and approaches to solving them.

We decided to focus on PB constraints because of our previous experience using them to generate timetables for one specific application, namely partial round-robin timetables for mahjong tournaments run in Europe (Lester 2021). We used the same approach: a monolithic encoding, followed by the ability to improve the timetable locally after opponent allocation was fixed. Mahjong is an unusual game from the perspective of sports timetabling because each game involves 4 competing players, rather than 2, and thus falls outside the scope of RobinX. In this case, generating a partial round-robin tournament timetable with no extra constraints is the (abstract) Social Golfer Problem (SGP) (Harvey 1999). In fact, the initial formulation of the SGP was in terms of PB constraints (Walser 1998). There have been several effective formulations of the SGP in SAT (Gent and Lynce 2005; Triska and Musliu 2012a; Lardeux and Monfroy 2015), but the best computational approach uses a heuristically guided tabu search (Triska and Musliu 2012b).

We are unaware of any other work on sports timetabling that directly targets the PB constraint format. On the other hand, there is plenty of work discussing use of 0–1 integer linear programming (ILP). For example, Ball and Webster (1977) discussed formulation of round-robin tournament timetabling as a 0–1 ILP. However, this is of limited relevance, as 0–1 ILP solvers use different techniques to PB solvers, and work better with different problem encodings.

Many constraint-based approaches to sports timetabling decompose the problem into several smaller subproblems. This is often more tractable than considering all constraints in the whole problem simultaneously. For example, Trick (2000) suggests a 2-phase “schedule-then-break” decomposition: fix opponents in each slot, then decide home/away patterns. Here, a home/away pattern is a sequence describing only in which rounds a particular team plays at home and in which it plays away. A break is when a team plays two consecutive games at home or two consecutive games away; the number of breaks a team has depends only on its home/away pattern. As breaks can influence a team’s performance, solutions to sports timetabling problems often seek to minimise the number of breaks. Trick’s approach has some similarities with our approach of using a monolithic encoding, then locally improving home/away pattern. However, our approach allows the home/away pattern to be considered in both phases.

Conversely, Henz uses a 3-phase decomposition in his general-purpose sports timetabling tool Friar Tuck (Henz 1999, 2001): generate home/away patterns; generate home/away pattern sets; and generate a timetable. In an early application of SAT to sports timetabling, Zhang (2002) uses the same 3-phase decomposition.

A disadvantage to the decomposition approach is that feasible solutions may be eliminated in each step, meaning that the optimal solution to the final subproblem, if one exists, is no longer necessarily an optimal solution to the original problem. If the solution found is not satisfactory, the implementation may have to restart or backtrack. In later work, Zhang et al. (2004) conclude that, if a timetabling tool is to be used non-interactively, a monolithic approach may be preferable. They develop a timetabling tool for a SAT solver extended to deal with cardinality constraints, which are a subset of PB constraints. It was available through a Web interface, which allowed the user to choose between single, double or partial double round-robin tournament and a combination of 9 different constraints on home/away patterns. Unfortunately, the Web page is no longer available.

One of the harder constraints in round-robin timetable generation, as discussed extensively by Rasmussen and Trick (2008), is break minimisation. For common tournament formats, there are well-known combinatorial designs that minimise breaks, but these often cannot be used in the presence of the complex combination of constraints that occurs in real-world timetabling problems. Indeed, the addition of these constraints often makes timetabling problems NP-complete. Many NP-complete problems are solvable in practice using SAT solvers, so this is a reasonable approach to try. Horbach et al. (2012) create a timetabling tool that accommodates a range of user-specified hard and soft constraints and solves them using a SAT solver. They handle soft constraints by adding them incrementally as necessary when a solution does not meet the required bound. They do not explain why they did not use an optimising MaxSAT or PB solver.

Horbach et al. (2012) observe that SAT solvers often perform badly at pigeonhole-type problems, of which constrained round-robin timetable generation is an example. This weakness can often be ameliorated through symmetry
breaking. The tool BreakID (Devriendt et al. 2016) supports symmetry breaking for SAT and PBS instances (although not PBO/WBO), but is unlikely to be helpful in practice, as the extra constraints in timetabling problems usually remove the symmetry that it would break.

3 Methods

3.1 Monolithic encoding

The ITC 2021 considered only time-constrained double round-robin tournaments (2RR) for an even number of teams. Each problem instance required creation of a timetable for \( n \) teams over \( 2(n-1) = 2n - 2 \) time slots, with each team playing each opponent exactly twice: once at home and once away. Some problems required a phased schedule, meaning that, for each pair of teams, the two games between them must be in different halves of the schedule. Additionally, each problem instance specified a range of other constraints. In the first instance, our tool solves the problems using a monolithic encoding with PB constraints. We now describe this encoding.

We number the teams 0 to \( n - 1 \) and the slots 0 to \( 2n - 3 \). We use the indices \( t, t_1 \) and \( t_2 \) to range over team numbers, \( s \) to range over slot numbers and \( h \) to range over \( \{0, 1, 2\} \). Where not otherwise specified, quantification of these indices is implicitly over these ranges, with \( t_1 \neq t_2 \). Our encoding uses the following sets of Boolean variables:

1. \( M_{t_1, t_2, s} \)—true just if team \( t_1 \) plays home against team \( t_2 \) in slot \( s \);
2. \( H_{t, s} \)—true just if team \( t \) plays home in slot \( s \);
3. \( B_{t, s, h} \)—true if team \( t \) has a home break (\( h = 1 \)) or an away break (\( h = 0 \)) in slot \( s \), with \( s > 0 \).

The timetable is determined entirely by the \( M \) variables. The remaining variables are auxiliary variables used to make expressing the constraints easier.

We generate some feasibility clause sets for all instances. Each team plays exactly once in each slot:

\[
\forall s, t_1, t_2, \sum_{t_2} (M_{t_1, t_2, s} + M_{t_2, t_1, s}) = 1
\]

Each home/away matchup between two teams occurs exactly once:

\[
\forall t_1, t_2, \sum_s M_{t_1, t_2, s} = 1
\]

(For phased schedules only:) Each pair of teams plays exactly once in each half:

\[
\forall t_1, t_2, \sum_{s \in [0, n-2]} M_{t_1, t_2, s} + M_{t_2, t_1, s} = 1
\]

\[
\forall t_1, t_2, \sum_{s \in [n-1, 2n-3]} M_{t_1, t_2, s} + M_{t_2, t_1, s} = 1
\]

Only one of these sets of constraints is necessary, as the other is then implied by the requirement that each matchup occurs exactly once. However, it is beneficial to include both sets, as this enables solvers to spot conflicts more quickly.

The home/away variables must reflect the choice of matches. If a team plays a specific home match in a slot, then it plays at home in that slot; if a team plays a specific away match in a slot, then it plays away in that slot:

\[
\forall s, t_1, t_2, -M_{t_1, t_2, s} + H_{t_1, s} \geq 0
\]

\[
\forall s, t_1, t_2, -M_{t_1, t_2, s} - H_{t_2, s} \geq -1
\]

The break variables must be true when a team has a home/away break in a slot. They need not be false when a team has no break, as none of the constraints we consider places a lower bound on the number of breaks permitted, although we may wish to add these constraints too, as we discuss later.

\[
\forall s > 0, t, B_{t, s, 1} + H_{t, s-1} + H_{t, s} \geq -1
\]

\[
\forall s > 0, t, B_{t, s, 0} + H_{t, s-1} + H_{t, s} \geq 1
\]

Next, we generate sets of constraints for each constraint in the problem instance as shown in Fig. 1.

3.2 Soft constraints

In many problem instances, some of the constraints are soft, meaning that they can be violated, but there is a penalty or cost for doing so. Furthermore, the cost varies according to how badly the constraint is violated. While the WBO format we used to encode each problem supports soft constraints with different weights, it does not directly support weights that vary according to the degree of violation. But in most cases, the variable weight could be encoded relatively simply.

Consider, for example, a CA1 home constraint encoded as:

\[
\sum_{s \in S} -H_{t, s} \geq -\max
\]

If the constraint is violated, the deviation \( d \) is:

\[
\left(\sum_{s \in S} H_{t, s}\right) - \max
\]
CA1: Team $t$ plays at most max home/away games in slots $S$:

\[
\text{Home: } \sum_{s \in S} -H_{t,s} \geq -\text{max} \\
\text{Away: } \sum_{s \in S} H_{t,s} \geq |S| - \text{max}
\]

CA2: Team $t_1$ plays at most max home/away/any games against teams in $T$ in slots $S$:

\[
\text{Home: } \sum_{s \in S, t_{2} \in T} -M_{t_{1},t_{2},s} \geq -\text{max} \\
\text{Away: } \sum_{s \in S, t_{2} \in T} -M_{t_{2},t_{1},s} \geq -\text{max} \\
\text{Any: } \sum_{s \in S, t_{2} \in T} -M_{t_{1},t_{2},s} - M_{t_{2},t_{1},s} \geq -\text{max}
\]

CA3: Teams in $T_1$ play at most max home/away/any games against teams in $T_2$ in each block of $p$ consecutive slots:

\[
\text{Home: } \forall t_{1} \in T_{1}, \forall s_{0} \in [0,|S| - p]. \sum_{s \in [s_{0},s_{0} + p - 1], t_{2} \in T_{2}\setminus \{t_{1}\}} -M_{t_{1},t_{2},s} \geq -\text{max} \\
\text{Away: } \forall t_{1} \in T_{1}, \forall s_{0} \in [0,|S| - p]. \sum_{s \in [s_{0},s_{0} + p - 1], t_{2} \in T_{2}\setminus \{t_{1}\}} -M_{t_{2},t_{1},s} \geq -\text{max} \\
\text{Any: } \forall t_{1} \in T_{1}, \forall s_{0} \in [0,|S| - p]. \sum_{s \in [s_{0},s_{0} + p - 1], t_{2} \in T_{2}\setminus \{t_{1}\}} -M_{t_{1},t_{2},s} + M_{t_{2},t_{1},s} \geq -\text{max}
\]

CA4 (global): Teams in $T_1$ play at most max home/away/any games against teams in $T_2$ over all slots in $S$:

\[
\text{Home: } \sum_{s \in S, t_{1} \in T_{1}, t_{2} \in T_{2}\setminus \{t_{1}\}} -M_{t_{1},t_{2},s} \geq -\text{max} \\
\text{Away: } \sum_{s \in S, t_{1} \in T_{1}, t_{2} \in T_{2}\setminus \{t_{1}\}} -M_{t_{2},t_{1},s} \geq -\text{max} \\
\text{Any: } \sum_{s \in S, t_{1} \in T_{1}, t_{2} \in T_{2}\setminus \{t_{1}\}} -M_{t_{1},t_{2},s} + M_{t_{2},t_{1},s} \geq -\text{max}
\]

CA4 (every): Teams in $T_1$ play at most max home/away/any games against teams in $T_2$ in each slot in $S$:

\[
\text{Home: } \forall s \in S. \sum_{t_{1} \in T_{1}, t_{2} \in T_{2}\setminus \{t_{1}\}} -M_{t_{1},t_{2},s} \geq -\text{max} \\
\text{Away: } \forall s \in S. \sum_{t_{1} \in T_{1}, t_{2} \in T_{2}\setminus \{t_{1}\}} -M_{t_{2},t_{1},s} \geq -\text{max} \\
\text{Any: } \forall s \in S. \sum_{t_{1} \in T_{1}, t_{2} \in T_{2}\setminus \{t_{1}\}} -M_{t_{1},t_{2},s} + M_{t_{2},t_{1},s} \geq -\text{max}
\]

GA1: Between min and max fixtures $(t_1, t_2)$ from $F$ (where $t_1$ plays $t_2$, with $t_1$ at home) occur in slots $S$:

\[
\sum_{s \in S, (t_1,t_2) \in F} M_{t_1,t_2,s} \geq \text{min} \\
\sum_{s \in S, (t_1,t_2) \in F} -M_{t_1,t_2,s} \geq -\text{max}
\]

BR1: Team $t$ has at most $p$ home/away/any breaks in slots $S$:

\[
\text{Home: } \sum_{s \in S\setminus \{0\}} -B_{t,s,1} \geq -p \\
\text{Away: } \sum_{s \in S\setminus \{0\}} -B_{t,s,0} \geq -p \\
\text{Any: } \sum_{s \in S\setminus \{0\}} -B_{t,s,0} + B_{t,s,1} \geq -p
\]

BR2: Teams in $T$ have at most $p$ breaks in slots $S$:

\[
\sum_{s \in S\setminus \{0\}, t \in T} -B_{t,s,0} + B_{t,s,1} \geq -p
\]

FA2: Each team in $T$ has played at most $p$ home games more than any other team in $T$ after slots in $S$:

\[
\forall t_{1} \in T. \forall t_{2} \in T : t_{1} < t_{2}. \forall s_{1} \in S. \sum_{s \in [0,s_{1}]} H_{t_{1},s} + -H_{t_{2},s} \geq -p \\
\forall t_{1} \in T. \forall t_{2} \in T : t_{1} < t_{2}. \forall s_{1} \in S. \sum_{s \in [0,s_{1}]} H_{t_{2},s} + -H_{t_{1},s} \geq -p
\]

SE1: Each pair of teams in $T$ has at least min clear slots between their mutual games:

\[
\forall t_{1} \in T. \forall t_{2} \in T - \{t_{1}\}. \forall s_{1} \in [0,2n - 4]. \forall s_{2} \in [s_{1} + 1, s_{1} + \text{min}] : s_{2} < 2n - 2. \\
-M_{t_{1},t_{2},s_{1}} + M_{t_{2},t_{1},s_{2}} \geq -1
\]
and the cost is $d \cdot w$, where the constraint specifies $w$. The maximum deviation, which we call $d_{\text{max}}$, is $|S| - \text{max}$. We can express this as a soft constraint by changing the original encoding (which remains a hard constraint) to:

$$\sum_{i \in [1, d_{\text{max}}]} D_i + \sum_{s \in S} -H_{t,s} \geq -\text{max}$$

where $D_1, \ldots, D_{d_{\text{max}}}$ are fresh variables, and adding a separate soft constraint $[w] - D_i \geq 0$ for each $i \in [1, d_{\text{max}}]$. Then, we can always satisfy the hard constraint by setting the $D_i$ variables to be true, but we pay a cost unit each time we do so. In order to break symmetry in setting these deviation variables, we can add the following clauses, which force them to be set monotonically:

$$\forall i \in [2, d_{\text{max}}], -D_i + D_{i-1} \geq 0$$

If the size of deviations were large, it might be more effective to use a binary encoding of deviation, where $\lceil \log_2 d_{\text{max}} \rceil$ variables are introduced with cost $2^w w$. However, for the ITC instances, the unary encoding we used seemed to work adequately.

To get the encoding of deviation correct, one must calculate the maximum possible deviation and generate the corresponding number of fresh deviation variables. Once this is done, this method is suitable for soft constraints of all types except FA2 and SE1.

For FA2, the deviation is calculated for each pair of teams, but we have separate clauses for $t_1$ playing more home games and for $t_2$ playing more home games. Therefore, in each pair of clauses, we must use the same deviation variables. For SE1, we have a separate clause for each violating inadequate separation, so we set the cost of deviation directly on each clause instead of introducing extra variables.

### 3.3 Variations in encoding

The ability of a SAT or PB solver to solve a problem can be sensitive to exactly how it is encoded. We now consider some variations on our encoding, which are implemented as options in Reprobate. For each option, we give the command-line switch that forces its use. As we will show in Sect. 5.3, each variation improves Reprobate’s performance on some instances, but makes it worse on others.

We have already mentioned the possibility of adding extra clauses to force break variables $B$ to be false when a team does not have a break (—break—sym); this made little difference to most ITC instances. We also described breaking symmetry in the deviation variables by enforcing monotonicity (—monotone); this can make a big difference in either direction.

The SE1 constraints could potentially be large if the required separation $\text{min}$ were large. To counteract this, we can introduce variables $S_{t_1, t_2, s}$, initially false, which flip to true for the slot where $t_1$ and $t_2$ first play, then flip back to false in the slot where they play a second time. This makes the SE1 constraint a simple cardinality constraint on the number of true $S$ variables, as well as making the deviation expressible using the same scheme as for most other constraints (—sep—count). However, we did not find that it led to any significant improvements, perhaps because $\text{min}$ is bounded by the number of rounds and hence is small in all ITC instances.

Our monolithic encoding was relatively weak at dealing with constraints on breaks. One variation that made some improvement here was introducing separate variables $B_{t,s,2}$ to indicate team $t$ had a break of either kind in slot $s$ (—ha—break):

$$\forall s > 0, t. B_{t,s,2} + -H_{t,s-1} + -H_{t,s} \geq -1$$
$$\forall s > 0, t. B_{t,s,2} + H_{t,s-1} + H_{t,s} \geq 1$$

Then, in the encoding of BR1 and BR2 constraints, the term $-B_{t,s,0} + -B_{t,s,1}$ could be replaced with $-B_{t,s,2}$.

The use of —ha—break had the greatest impact when combined with an idea taken from Horbach et al. (2012). We can introduce variables $P_s$ to track break periods. $P_s$ is true just if at least one team has a break in slot $s$:

$$\forall s > 0, t. -B_{t,s,2} + P_s \geq 0$$
$$\forall s > 0, -P_s + \sum_t B_{t,s,2} \geq 0$$

Furthermore, there may never be 3 consecutive break periods (—triple):

$$\forall s \in [1, 2n - 5], -P_s + -P_{s+1} + -P_{s+2} \geq -2$$

This constraint is not sound in general; it may conflict with some constraints specified in a problem instance. However, it is true for otherwise unconstrained round-robin timetables with the minimal number of breaks and for many other timetables with a small number of breaks. Thus it is useful for instances with a large BR2 constraint that restricts the total number of breaks in a timetable, as it gives the solver some local information about where breaks should occur.

Finally, it is possible to omit all soft constraints entirely and encode only the hard constraints (—hard). This may help some solvers that fail to find a feasible solution because they are distracted by attempting to satisfy a large number of soft constraints.
3.4 Local improvement

When Reprobrate finds a solution using the monolithic encoding, it will not necessarily be optimal. To improve upon this, it can attempt to improve the timetable using a second encoding, in which the opponent of each team in each slot is fixed according to the initial timetable, but the home/away pattern may change.

Now the $H$ variables become the decision variables. Our goal is to remove the $M$ variables from the problem encoding. We add extra clauses to express that, for any pair of teams, their home/away status must swap between their games, and when one team is at home, the other must be away. For any teams $t_1$ and $t_2$, let $S_1(t_1, t_2)$ be the slot containing the first game between $t_1$ and $t_2$ in the initial timetable; correspondingly, let $S_2(t_1, t_2)$ be the second. Then:

\[
\forall t_1, t_2. H_{t_1, S_1(t_1, t_2)} + H_{t_1, S_2(t_1, t_2)} = 1
\]

\[
\forall t_1, t_2 \geq t_1. H_{t_1, S_1(t_1, t_2)} + H_{t_2, S_1(t_1, t_2)} = 1
\]

\[
\forall t_1, t_2 > t_1. H_{t_1, S_2(t_1, t_2)} + H_{t_2, S_2(t_1, t_2)} = 1
\]

Note that fixing the home/away status of a team in a time slot also determines the home/away status of the opposing team and the home/away status of the return game, so only 1/4 of the $H$ variables serve as decision variables; the rest are auxiliary.

We can substitute 0 or $H_{t_1, s}$ for each $M_{t_1, t_2, s}$ in the original encoding, depending on the initial timetable. Of the feasibility clauses used for all instances, those that referred to $M$ variables become redundant and can be discarded; only the clauses linking $B$ and $H$ variables remain.

Constraints of type CA1, BR1, BR2 and FA2 only refer to home/away pattern, so are unchanged. As the slots containing matches between teams are fixed, constraints of type SE1 cannot be affected by altering home/away pattern and so can be removed. This is also true for constraints of type CA2, CA3 or CA4 that refer to games of any type, rather than specifically home or away games. Constraints of type GA1 need to be modified, as do constraints of type CA2, CA3 or CA4, if they refer specifically to home or away games.

The modified constraints are as in Fig. 2. We define $O(t_1, s)$ to be the opponent of team $t_1$ in slot $s$; that is, $O(t_1, s) = t_2$ if, in the solution to the monolithic encoding, either $M_{t_1, t_2, s}$ or $M_{t_2, t_1, s}$. We write $O(t_1, S)$ to mean the image $\{t_2 | \exists s \in S. O(t_1, s) = t_2\}$. This is used in constraints on the number of away games to express the number of relevant games that might be at home; subtracting the number of home games from this gives the number of away games.

Fixing the allocation of opponents significantly simplifies many of the constraints, so it is reasonable to suppose that a better solution might be found this way, even though it would have been a valid solution to the initial encoding. The optimal solution to the instance may not be possible within the second encoding, but the first solution always remains possible.

3.5 Beyond ITC 2021

At present, Reprobrate only handles the subset of the RobinX timetabling format (Van Bulck et al. 2020b) considered in the ITC 2021. That is, it only considers 2RR tournaments where the objective is minimisation of sum of violated soft constraints and only a subset of constraint types are supported. However, the tool could be extended to support all constraint types, with varying levels of ease and performance.

Of the unimplemented constraint types, GA2, SE2 and FA3 are straightforward SAT-style constraints that are easily expressed in a PB encoding.

Some constraint types, like CA5, FA1, BR3 and BR4, involve some kind of counting of slots or comparison of numbers of breaks. These are slightly harder to encode, but are essentially cardinality constraints, which are also easy to express in a PB encoding. As with the already implemented constraints FA2 and SE1, if the sizes of sets of teams or slots involved in these constraints are large, the encoding will also be large, and performance may suffer.

The RobinX constraint types most different from those in the ITC 2021 are FA4, FA5 and FA6, all of which involve addition of costs that, unlike a team’s number of games played, need not be small integers. For example, FA5 concerns the sum of distances travelled by teams, and can be used to express instances of the travelling tournament problem (TTP). Expressing the similar travelling salesman problem (TSP) in SAT usually involves encoding binary adder circuits to sum the costs travelled (Zhou et al. 2015). However, there is a more direct encoding of TSP/TTP into PB constraints, where the travel costs map directly onto coefficients of variables (Manquinho and Roussel 2006). This approach could be adapted to these constraint types, although neither SAT nor PB encodings of TSP/TTP are very competitive in practice.

4 Implementation details

Our tool, Reprobrate, is implemented as a collection of Perl scripts, which call existing PB solvers such as clasp. The leading single-algorithm SAT and PB solvers have different strengths and weaknesses, with regard to the problems they can solve. The best practical SAT solvers are portfolio solvers. They run a number of single-algorithm solvers sequentially or in parallel, perhaps with different timeouts. Reprobrate uses a naive portfolio on the default problem encoding: it runs two solvers (clasp and Sat4J) with a small range of common settings and the same timeout and returns
CA2*: Team \( t_1 \) plays at most \( \text{max} \) home/away games against teams in \( T \) in slots \( S \):

\[
\begin{align*}
\text{Home} : & \sum_{s \in S, O(t_1, s) \in T} -H_{t_1, s} \geq -\text{max} \\
\text{Away} : & \sum_{s \in S, O(t_1, s) \in T} H_{t_1, s} \geq |O(t_1, S) \cap T| - \text{max}
\end{align*}
\]

CA3*: Teams in \( T_1 \) play at most \( \text{max} \) home/away games against teams in \( T_2 \) in each block of \( p \) consecutive slots:

\[
\begin{align*}
\text{Home} : & \forall t_1 \in T_1, \forall s_0 \in [0, |S| - p], \sum_{s \in [s_0, s_0 + p - 1], O(t_1, s) \in T_2} -H_{t_1, s} \geq -\text{max} \\
\text{Away} : & \forall t_1 \in T_1, \forall s_0 \in [0, |S| - p], \sum_{s \in [s_0, s_0 + p - 1], O(t_1, s) \in T_2} H_{t_1, s} \geq |O(t_1, [s_0, s_0 + p - 1]) \cap T_2| - \text{max}
\end{align*}
\]

CA4’ (global): Teams in \( T_1 \) play at most \( \text{max} \) home/away games against teams in \( T_2 \) over all slots in \( S \):

\[
\begin{align*}
\text{Home} : & \sum_{s \in S, t_1 \in T_1, O(t_1, s) \in T_2} -H_{t_1, s} \geq -\text{max} \\
\text{Away} : & \sum_{s \in S, t_1 \in T_1, O(t_1, s) \in T_2} H_{t_1, s} \geq |\sum_{t_1 \in T_1} |O(t_1, S) \cap T_2|| - \text{max}
\end{align*}
\]

CA4’ (every): Teams in \( T_1 \) play at most \( \text{max} \) home/away games against teams in \( T_2 \) in each slot in \( S \):

\[
\begin{align*}
\text{Home} : & \forall s \in S, \sum_{t_1 \in T_1, O(t_1, s) \in T_2} -H_{t_1, s} \geq -\text{max} \\
\text{Away} : & \forall s \in S, \sum_{t_1 \in T_1, O(t_1, s) \in T_2} H_{t_1, s} \geq |\sum_{t_1 \in T_1} |O(t_1, \{s\}) \cap T_2|| - \text{max}
\end{align*}
\]

GA1*: Between \( \text{min} \) and \( \text{max} \) games from \( F \) occur in slots \( S \):

\[
\begin{align*}
\sum_{s \in S, (t_1, t_2) \in F: O(t_1, s) = t_2} H_{t_1, s} & \geq \text{min} \\
\sum_{s \in S, (t_1, t_2) \in F: O(t_1, s) = t_2} -H_{t_1, s} & \geq -\text{max}
\end{align*}
\]

Fig. 2 Encodings of instance-specific constraints for local improvement

the best solution found by any of them. In a similar vein, as *Reprobate* supports some variations in the encoding, it tries several combinations and returns the best solution found with any of them. In order to keep running time reasonable, by default *Reprobate* only uses a single solver for the improvement process and for variations on the default encoding.

Our choice of solvers was somewhat limited, as there are few solvers under active development that support the WBO format. For *clasp*, we used the default and *crafty* (combinatorially hard) presets. For *Sat4J*, we used the recent 2.3.6 release with the default algorithm as well as the new CuttingPlanes and RoundingSat algorithms and hybrid Partial algorithm. The only other current WBO solver we are aware of, *ToySolver*, performed poorly, so we did not use it. Other current OPB format solvers, such as *OpenWBO*, *RoundingSat*, *Exact* and *UWrMaxSat*, do not support WBO problems, although they can be used with *Reprobate* in combination with the \(--\text{hard}\) flag, which generates a PBS problem with no soft constraints and in OPB format.

As mentioned in Sect. 2.1, a linear PB instance can also be expressed as 0–1 integer linear programming instance, a special case of mixed integer programming (MIP). To investigate performance of MIP solvers, we modified *Reprobate* to output problem encodings in the widely supported MPS format for MIP. This required several adaptations. We began by converting soft clauses into an objective function. This involved two steps: firstly, we changed the encoding of non-unitary soft clauses from the SE2 constraints to use deviation variables instead; secondly, we set the objective function to be the sum of all deviation variables, appropriately weighted, and removed the corresponding unitary soft clauses. Then we output the clauses in CPLEX LP format, which is syntactically similar to OPB, before using the open source MIP solver GLPK to convert this into MPS format.

5 Results

5.1 Baseline monolithic encoding performance

In the ITC 2021, there were 45 problem instances. We ran *Reprobate* on these during the competition. While it did not place among the top half of the competition, it did manage to generate feasible solutions for 29 out of 45 instances (64%). Of these, 27 were generated with just 600 s of CPU time; many entrants to the competition used much more.
addition after the competition of the “no triple break period” constraint, we were able to solve 4 more, increasing that to 73%. According to the competition report (Van Bulck et al. 2021), “for most problem instances, a straightforward integer programming formulation could not even generate a feasible solution”, so Reprobate is superior to that.

Before we can consider the impact of variations in our constraint encoding and solving process, we need to establish a baseline for comparison. Initial experiments suggested using just clasp as the solver with the crafty preset (for combinatorially hard problems) and a timeout of 600 s was adequate for Reprobate to generate feasible solutions for most problems, so we adopt that as our baseline. Table 1 shows the ITC 2021 instances solved and their objective scores. All results were generated on a machine running Debian Linux 10 with a 3.4 GHz Intel Core i5-7500 CPU and 64 GB of RAM. Each solver was run on a single core.

### 5.2 Improving feasibility

Let us firstly consider adjustments that improved feasibility. The ITC did not specify any limits on computation time, and many entrants used much more than us. Adopting a portfolio of solvers allowed Reprobate to solve one more instance using Sat4J-partial: Early 06. Increasing the timeout for all solvers from 600 to 5000 s (as used in the SAT Competition 2020) enabled Sat4J-rounding to find 2 more solutions: Middle 14 and Late 12. Reverting to a timeout of 600 s and considering only hard constraints allowed Reprobate to solve another instance: Early 02. Finally, adopting the “no triple consecutive break period” constraint after the competition and using the cutting-edge PB solver Exact with hard constraints only enabled solution of 4 more instances: Early 01, Early 11, Early 13 and Late 7. All these instances are marked with an asterisk in Table 1. We also tried increasing the time for our default encoding and solver to 36,000 s, but this did not yield any new solutions.

Out of the instances that Reprobate could not solve during the competition, all except Middle 3 featured a large, hard BR2 constraint that put a bound on the number of permissible breaks over the whole timetable. We confirmed that this was the source of the problem by removing constraints from the instances, observing that an instance containing just the BR2 constraint was not solvable. This led us to implement the “no triple consecutive break period” constraint, which we evaluated using a sequence of artificially constructed instances for phased tournaments with 2–20 teams. Each instance had a BR2 constraint, restricting the number of breaks to 3(n − 2). This is relatively tight, as it is the minimum in a mirrored tournament, with the minimum in a phased tournament being 2(n − 2). With the above modification, Exact could solve the cases for 12 and 14 teams in 7 and 65 min, respectively, which was an improvement, but still a long way off being able to solve for 20 teams.

### 5.3 Improving objective

Now we consider how various factors affected the objective in the monolithic encoding. We focus our attention on the 25 instances solvable in the baseline case. Firstly, we look at how using a portfolio of solvers affects the objective. Figure 3 shows the relative increase or decrease in objective value attained, for each solver in our portfolio, compared with the baseline. (Each solver finds its own solution independently; it does not use the solution found by the baseline as a starting point.) The bottom point for each column is thus the value attained by using a portfolio of all solvers. Each solver performed best on at least one instance, but clasp (crafty) had the highest number of best solutions. Sat4J-partial and Sat4J-rounding were similar in terms of feasible solutions, with 25 and 24 instances solved, respectively, but with a lower number of best solutions among the portfolio. Sat4J-default was clearly the weakest, with only 6 feasible solutions, of which 1 was a best solution.

Next we consider the effect of variations in the encoding, while keeping the solver as clasp (crafty). Figure 4 shows how adopting each variation in isolation affects the objective. Each variation improves the objective for some instances and makes it worse for others. As with our choice of PB solver, we can adopt a portfolio approach to choice of variations. However, we have not benchmarked combinations of variations, which may well perform better than individual variations for some instances. The most dramatic improvements came from enforcing monotonicity of deviation variables and from forbidding triple consecutive break periods. This is perhaps to be expected, as the former breaks many symmetries, while the

| Table 1 Baseline performance of Reprobate on ITC 2021 instances |
|---------------------------------------------------------------|
|                | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 15 |
|-----------------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| Early           | *  | *  | 4884 | *  |    |    | 5584 | 4858 | *  | 2070 | *  | 3489 | 7443 |
| Middle          | 99 | 2901 | 3235 | 8563 |    |    | 1189 | 3530 | 4263 | 4950 | 6199 | *  | 7590 |
| Late            | 3683 | 7784 | 0  | 3678 | *  |    | 4583 | 2940 | *  | 8564 | 3712 | 5910 |

Figures show objective score with default encoding, clasp (crafty) PB solver, a timeout of 600 s and no local improvement phase. Lower is better. Instances marked with * can be solved using other settings.
latter directly addresses the biggest weakness of the monolithic encoding. However, these also produced some of the biggest regressions in performance, including instances that ceased to be feasible. This is less surprising for the restriction on break periods, as it is unsound. Also of note is that generating hard constraints only led to a better objective in some cases, suggesting that the PB solvers often struggle to make any improvement once they have found a feasible solution.

5.4 Comparison of PB and MIP solvers

Reprobate targets the WBO format, which allows us to use specialised PB solvers, but this raises the question of how well a MIP solver would perform on the equivalent formulation. Berthold et al. (2008) argue that “feasibility problems with many constraints that have 0/1 coefficients only” will most likely work best with PB solvers, while “instances with many inequalities with arbitrary coefficients” will work best with MIP solvers. Our encoding uses only $+/-1$ coefficients, so all constraints are either pure SAT constraints (as with the constraints linking the $M$, $H$ and $B$ variables) or cardinality constraints (which have efficient SAT encodings). Therefore, we would expect the PB solvers to perform better.

We ran a range of popular MIP solvers on the MPS encodings for the ITC instances. We tried: the open source solvers lp_solve, glpsol from GLPK, and CBC from COIN-OR; SCIP (source available, but free for academic use only); and the commercial solvers Gurobi and CPLEX. We used the same benchmark settings as our baseline: 600 s of CPU time on a single core. The non-commercial solvers failed to find feasible solutions for any instances in this time. Gurobi found 1 feasible solution (Late 15), while CPLEX found 6 (Early 3, 14; Late 3, 4, 8, 15). So with our baseline time limit, the best MIP solver was comparable to the worst PB solver in our portfolio.
The commercial solvers became more competitive when given 5000s of CPU time. Gurobi found 6 feasible solutions, and CPLEX found 18, but none were for previously unsolvable instances. Some of the objective scores were better than those we had previously found, so they might be worthwhile additions to a portfolio where licensing allows.
In any case, our results agree with the claims of Berthold et al. (2008).

5.5 Evaluation of local improvement

We now evaluate the effect of the local improvement process. For this, we used the monotonic encoding of deviation variables and clasp (crafty) with a timeout of 600 s. Figure 5 shows the improvement when applying local improvement just to the baseline and when applying it to the portfolio. The process almost always improves the objective. The decrease is usually less than 10%, but can be significantly more in some cases. Note that the local improvement process does not always find a solution as good as the original. In such cases, Reprobate reverts to using the original solution.

Overall, if we look at the best solution found for each instance, whether by using a portfolio of solvers with the original encoding, or by using clasp (crafty) on a variation, and whether locally improved or not, the average relative objective, compared with just using clasp (crafty) on the original encoding, is 77%. That is, our efforts to improve the objective yielded an average decrease of 23%. This is a significant improvement, although there is plenty of room for more: the points at the bottom of Fig. 5 show the best solutions submitted by any team to the ITC 2021. Table 2 shows the corresponding absolute numerical values.

6 Conclusion

We have developed and evaluated Reprobate, a tool that solves the subset of RobinX format sports timetabling problems considered in the International Timetabling Competition 2021. The primary technique used by our tool is a monolithic encoding using pseudo-Boolean constraints, which can be solved using existing solvers, such as clasp. This is augmented by a second local improvement step, which uses pseudo-Boolean constraints to adjust the home/away pattern. Our tool was effective on many of the problems in the ITC 2021, although it struggled with large break constraints. Both Reprobate and clasp are distributed under the open source MIT License, making our system readily available for others to use or improve. Our work reaffirms the message that pseudo-Boolean constraints are a powerful and expressive formalism for modelling many real-world problems, for which high-quality off-the-shelf solvers are available. It also demonstrates the value of using a portfolio of solvers, rather than relying on a single good solver. However, more work is needed to understand how best to encode and solve break minimisation constraints using a SAT or PB solver. There is also scope for extending Reprobate to handle those RobinX constraints and tournament formats that were not considered in the ITC 2021.

Data Availability Statement Source code and data supporting the results in this article are available from a repository (Lester 2021) hosted on Zenodo.

Declarations

Conflict of interest The author has no relevant financial or non-financial interests to disclose.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Bacchus, F., Järvisalo, M., & Martins, R. (2019). MaxSAT evaluation 2018: New developments and detailed results. Journal on Satisfiability, Boolean Modeling and Computation, 11(1), 99–131. https://doi.org/10.3233/SAT190119.

Ball, B. C., & Webster, D. B. (1977). Optimal scheduling for even-numbered team athletic conferences. AIIE Transactions, 9(2), 161–169. https://doi.org/10.1080/05695557708975138.

Berre, D. L., & Parrain, A. (2010). The Sat4j library, release 2.2. Journal on Satisfiability, Boolean Modeling and Computation, 7(2–3), 59–64. https://doi.org/10.3233/sat100075.

Berthold, T., Heinz, S., & Pletsch, M. (2008). Solving pseudo-Boolean problems with SCIP. Technical Report 08-12, ZIB, Takustr. 7, 14195 Berlin. https://nbn-resolving.org/urn:nbn:de:0297-zib-10671.

Devriendt, J., Bogaerts, B., Bruynooghe, M., & Denecker, M. (2016). Improved static symmetry breaking for SAT. In N. Creignou, & D. L. Berre (Eds.), Theory and applications of satisfiability testing—SAT 2016–19th international conference, Bordeaux, France, July 5–8, 2016, proceedings. Lecture notes in computer science (Vol. 9710, pp. 104–122). Springer. https://doi.org/10.1007/978-3-319-40970-2_8.

Devriendt, J., Gleixner, A. M., & Nordström, J. (2021). Learn to relax: Integrating 0–1 integer linear programming with pseudo-Boolean conflict-driven search. Constraints, 26(1), 26–55. https://doi.org/10.1007/s10601-020-09318-x.

Eén, N., & Sörensson, N. (2006). Translating pseudo-Boolean constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation, 2(1–4), 1–26. https://doi.org/10.3233/sat190014.

Elflers, J., & Nordström, J. (2018). Divide and conquer: Towards faster pseudo-Boolean solving. In J. Lang (Ed.), Proceedings of the twenty-seventh international joint conference on artificial intelligence, IJCAI 2018. July 13–19, 2018, Stockholm, Sweden (pp. 1291–1299). ijcai.org. https://doi.org/10.24963/ijcai.2018/180.
