A SAT Based Scheduler for Tournament Schedules*

Hantao Zhang, Dapeng Li, and Haiou Shen

Department of Computer Science
The University of Iowa
Iowa City, IA 52242-1419, USA
{*dapli, hshen, hzhang*}*@cs.uiowa.edu*

Abstract. We present a scheduler with a web interface for generating fair game schedules of a tournament. The tournament can be either single or double round-robin or something in between. The search engine inside the scheduler is a SAT solver which can handle a mix of ordinary and TL (True-Literal) clauses. The latter are the formulas using the function TL which counts the number of true literals in a clause. By using TL clauses, we could solve a typical scheduling problem in a few of seconds. If we convert them into ordinary clauses, the state-of-the-art SAT solvers could not solve them in one week. We showed how to integrate TL clauses into a SAT solver and take advantages of the advanced SAT techniques. Our scheduler provides a free service to all the people who are interested in fair sports schedules.

1 Introduction

In recent years, the scheduling of sport leagues has become an important application area of combinatorial search techniques. For instance, Schaerf [11] used constraint logic programming to generate schedules for Dutch "Top League" and Italian "Series A" of football. Henz [7] employed finite-domain constraint programming to generate an ACC (Atlantic Coast Conference) basketball schedule. Hamiez and Hao [6] used tabu search to solve a specific sports league scheduling problem. Zhang [19] used a SAT solver to solve a BigTen basketball scheduling problem. These methods can solve sports scheduling problems with a variety of success.

Major college conferences basketball matches draw millions of viewers each year. Besides team members care about the fairness of a schedule, there are also loyal fans and die-hard team followers who are upset with inconvenient schedules, not to mention leaders of other sports, such as swimming and wresting, who have to schedule around basketball games. Since weekend home games draw larger audiences, and weekday road homes may hurt players' study, for a schedule to be fair, we may require that each team plays exactly the same number of home games on weekends and on weekdays and the same number of road games on weekends and on weekdays. According to this criterion, many schools in the states, including several BigTen schools, do not have a fair schedule in the current season [16].

After we successfully generated fair schedules for several college conferences with a SAT solver [19], we thought that it would be useful if we can build a scheduler based on our techniques and provide a web interface on the top of it so that everyone can use it. After some experiments, we found this is a very challenging task because the approach took in [19] is the same as in [10, 7] which divides the whole process into three phases: (a) pattern generation; (b) pattern set generation; and (c) schedule generation. In each of these phases, a different set of propositional clauses is created and a call to the SAT solver is made. After each run, its results are extracted and used for the next run. Because of various constraints and sizes of tournaments, some instances may produce too much intermediate results for a web server to handle. Moreover, because three runs of a SAT solver are needed, it is very difficult to generate the next solution if the user does not like the first solution.

Fortunately, there is a feature implemented in our SAT solver for many years and we decided to take advantage of this feature and specify the scheduling problem in a single SAT instance. To our surprise, the SAT solver could produce a schedule in one run in short time. The special feature of our solver is that we can specify the constraints of a problem using the function TL

* Supported in part by NSF under grant CCR-0098093.

(True-Literal) of clauses: For each clause c, TL(c) returns the number of true literals under the current assignment. If c contains literals $l_1, l_2, ..., l_n$ and each literal is regarded as a 0-1 variable, then $\mathrm{TL}(c) = l_1 + l_2 + \cdots + l_n$. It turns out that by using TL, we reduce substantially the size of the SAT instance from the scheduling problem. Without using TL, we could not solve a problem after running the SAT solver for over a week; using TL, the same problem can be solved in less than one second.

2 The Scheduling Problem as Constraint Satisfaction

To formulate the tournament schedule problem as a constraint satisfaction problem, let us denote the teams by 1 through n, and number the game days 1 through m, and define a set of 0/1 variables $p_{x,y,z}$, where $1 \le x \le n, 1 \le y \le n$, and $1 \le z \le m$: $p_{x,y,z} = 1$ if and only if x plays a home game against y in game day z.

Various constraints can be expressed formally using the variables $p_{x,y,z}$. The following are a few of examples.

1. Each team can play at most once in one game day: For $1 \le x \le n$, $1 \le z \le m$,
$$\sum_{y=1}^{n}(p_{x,y,z} + p_{y,x,z}) \le 1.$$
2. Each team must play each other team at least once and no more than twice: For $1 \le x, y \le n$, $x \ne y$,
$$\sum_{z=1}^{m} p_{x,x,z} = 0, \quad \sum_{z=1}^{m} p_{x,y,z} \le 1, \quad \sum_{z=1}^{m} p_{y,x,z} \le 1, \quad 1 \le \sum_{z=1}^{m}(p_{x,y,z} + p_{y,x,z}) \le 2;$$
3. Each team plays k home and k road games in weekends and in weekdays: Suppose an even numbered game day is a weekend and an odd numbered game day is a weekday, then for $1 \le x \le n$,
$$\begin{array}{ll} \sum_{y=1}^{n}\sum_{i=1}^{m/2} p_{x,y,2i-1} = k, & \sum_{y=1}^{n}\sum_{i=1}^{m/2} p_{x,y,2i} = k, \\ \sum_{y=1}^{n}\sum_{i=1}^{m/2} p_{y,x,2i-1} = k, & \sum_{y=1}^{n}\sum_{i=1}^{m/2} p_{y,x,2i} = k; \end{array}$$
4. No three home games in consecutive days and no three road games in consecutive days: For $1 \le x \le n$, $1 \le z \le m-2$,
$$\sum_{y=1}^{n}\sum_{i=0}^{2} p_{x,y,z+i} < 3, \quad \sum_{y=1}^{n}\sum_{i=0}^{2} p_{y,x,z+i} < 3.$$

Because the domains of this constraint problem are finite, theoretically, each of the above constraints can be converted into a set of propositional clauses, treating $p_{x,y,z}$ as propositional variables. Given a set S of n 0/1 variables, to specify $\sum_{x \in S} x = k, 1 \le k < n$, we may use the following propositional clauses: For any $X \subseteq S$, $|X| = 1 + k$, create the clause $\vee_{x \in X}\overline{x}$, where $\overline{x}$ is the negation of x; for any $Y \subseteq S$, $|Y| = n-k+1$, create the clause $\vee_{y \in Y} y$. The first type of clauses ensures that no more than n variables are true (1); the second type of clauses ensures that no more than $n-k$ variables are false (0). The number of clauses is thus $C_n^{k+1} + C_n^{n-k+1}$, or equivalently, $C_n^{k+1} + C_n^{k-1}$. Hence, it is very expensive to use these constraints in an ordinary SAT solver.

We are able to obtain a set of over 10 million propositional clauses (the number of propositional variables is 2178) from Constraints 1–4 for the BigTen conference basketball schedule, where $n = 11$ and $m = 18$. However, Sato [18, 21] cannot produce any solution from this set of clauses after seven days of running. It appears that this approach leads to a dead end.

The successful approach reported in the literature is to use a three-phases approach, namely pattern generation, pattern set generation, and timetable generation, to search for a schedule. In each phase of the three-phases approach, various constraints are used to narrow down the search space. This approach was proposed by Cain [3], and used by de Werra [15] and Schruder [12], among others. In particular, this approach has been taken by Nemhauser and Trick [10]

for the 1997/98 ACC basketball schedule. The ACC has 9 teams and plays a double round-robin tournament for the regular season basketball games. Each team plays 16 games (8 home games and 8 road games). Nemhauser and Trick's result was accepted by the ACC as their official basketball schedule [10]. They reported a "turn-around-time" of 24 hours using integer programming and explicit enumeration, which means it takes one day of computing time from specifying/modifying the constraints to proposing new solutions. This "turn-around-time" has been dramatically reduced to one minute by Henz [7], who used the finite-domain constraint programming [9, 14]. Zhang showed that SAT solvers can be used successfully in this approach. Not only solved the ACC problem with the "turn-around-time" of one second, Zhang also solved the BigTen basketball scheduling problem which is much more complex than the ACC problem, because the BigTen has 11 schools and the schedule is not a complete double round-robin tournament [19].

For the ACC scheduling problem [10, 7], Zhang could obtain all the 179 schedules in less than one second [19]. In comparison, Henz' Friar Tuck [8] took 53.7 seconds. This experiment shows clearly that SAT solvers provide an alternative and efficient tool for scheduling problems. The short "turn-around-time" of our solutions allows us to quickly pass through multiple cycles of problem refinement to satisfy all parties involved.

One major problem with the three-phase approach to a generic scheduler is that the number of pattern sets grows exponentially with the number of patterns, and the number of timetables grows exponentially with the number of pattern sets. If the number of teams is big or the desired constraints are few, then a big number of patterns will be generated. This will produce a huge number of pattern sets and the scheduler will be stuck on this job. From our experiences, if the number of patterns is greater than 100, then the scheduler would fail to generate a schedule in a reasonable amount of the time.

3 Handling TL Constraints in a SAT Solver

We have seen that the basketball tournament schedule problem can be easily stated as a constraint satisfaction problem. Even if we have the most advanced computer technology at hand, if we are not careful on how to specify the problem properly, the solution can still be intractable. In section 2, we saw that for the constraint like $\sum_{x \in S} x = k, 1 \leq k < n = |S|$, will generate $C_n^{k+1} + C_n^{k-1}$ clauses. However, if we treat S as a single clause (i.e., a list of literals), then the same constraint can be expressed by a single formula $\mathrm{TL}(S) = k$. In fact, most constraints in the scheduling problem can be specified using TL, thus greatly reducing the number of formulas required to represent these constraints.

The key idea is to impose an additional constraint on a clause that at most k literals in it can be true based on a given formulation. A clause with such a constraint is called a TL-clause; k is different for different TL-clauses. This information is kept in a counter, which is initialized to be k and is decremented every time a literal in the clause is assigned true during the propagation of boolean assignments.

The idea of using non-CNF formulas in a SAT solver is not new [1, 2]. For instance, Aloul et al. [1] combined a specialized 0-1 linear programming solver with a SAT solver and reported great performance improvements for many application problems. The constraints in their ILP solver are of the form $\sum_{i=1}^{n} a_i x_i \leq b$, where a_i, b are integers and x_i are 0-1 variables. If for every i, $a_i = 1$, then the same constraint can be expressed as $\mathrm{TL}(x_1 \vee x_2 \vee \cdots \vee x_n) \leq b$. In other words, the TL constraints considered here are a special case of their constraints.

The main reason we consider only the TL constrains (called TL-clauses from now on) instead of general 0-1 constraints because we want integrate TL-clauses seamlessly into a SAT solver, in order to fully take advantages of the advanced techniques developed for SAT solvers. In other words, we want to take advantage of the clause structure and, at the same, provide flexibility for a user to specify constraints efficiently.

For instance, all the advanced SAT solvers today accept clauses in the DIMACS CNF format. In this format, a literal is represented by a non-zero integer and a clause is represented by a list of integers followed by a zero. We extended the DIMACS CNF format to represent TL-clauses as follows: A TL-clause is a list of integers followed by one symbol in $\{<, <=, >=, >, =\}$ and a natural number. TL clauses and ordinary clauses can be mixed in any order in the input. In preprocessing,

a TL clause is converted into one or two TL clauses of the normal form $\mathrm{TL}(c) \leq k$, where k is recorded in the clause.

Not only TL clauses share the input format with ordinary clauses, TL clauses can also share the data structures with ordinary clauses. To satisfy a TL clause like $\mathrm{TL}(c) \leq k$, the number of true literals in S cannot be more than k. To check this satisfiability, we add a counter associated with a TL clause c, called TLcount(c), which is initialized to be k.

During the propagation of the boolean assignments, when a literal in c becomes true, the TLcount(c) is decreased by one. If TLcount(c) ≤ 0, the clause c is scanned to make sure that the value of TLcount(c) is correct (if not, the old value is discarded). If TLcount(c) is indeed 0, then the propagation continues and every unassigned literal in c will be assigned to be false; if TLcount(c) is less than 0, then the propagation terminates and a conflict is reported.

One of the advanced features for boolean propagation [22] is that the precondition of clauses for propagation is relaxed so that when backtracking we do not have to undo every operation performed in the propagation. The propagation procedure for TL clauses described above also has this feature: It takes constant time to backtrack. This idea can be used by SAT solvers where a counter is associated with each ordinary clause to record unassigned literals [5].

Another successful technique for the advanced SAT solvers is conflict-driven lemma learning [13, 18, 22]. TL clauses are used in our SAT solver as ordinary clauses for this purpose with only two exceptions: (a) the false literals in a TL clause are ignored and only true literals are collected for further consideration; (b) a TL clause may have caused more than one literal to be assigned false during the propagation. Since TL and ordinary clauses share data structures in our SAT solver, the change to the learning procedure is straightforward and we illustrate it by an example.

Example 1. Suppose one of TL clauses is $\mathrm{TL}(x \vee y \vee z \vee w) \leq 2$ and two binary clauses are $z \vee u$ and $\overline{u} \vee w$. During the search, after assigning $x = y = 1$, z and w are forced to be 0 by $\mathrm{TL}(x \vee y \vee z \vee w) \leq 2$. If $z = 0$ is treated first, then we have $u = 1$ from $(z \vee u)$ and $(\overline{u} \vee w)$ becomes a conflict [22]. Note that the *ancestor* of z and w is the same, i.e., $\mathrm{TL}(x \vee y \vee z \vee w) \leq 2$; the *ancestor* of u is $(z \vee u)$. To generate a lemma from the conflict, we trace the ancestors of each literal in the conflict clause. From u, we get $(z \vee u)$ and from $\overline{w}$, we get $\mathrm{TL}(x \vee y \vee z \vee w) \leq 2$. Since the latter is a TL clause, only the positively assigned literals in this TL clause (in this case, x and y) are collected. In other words, the *ancestor* of w is actually treated as $(\overline{x} \vee \overline{y} \vee \overline{w})$, which is one of those if we want to transform $\mathrm{TL}(x \vee y \vee z \vee w) \leq 2$ into ordinary clauses. Similarly, the *ancestor* of z is treated as $(\overline{x} \vee \overline{y} \vee \overline{z})$.

Using TL clauses, many interesting SAT related problems can be specified easily and efficiently. In fact, all the examples in [1] can be expressed as TL or ordinary clauses. The SAT solver supporting TL clauses outperforms Aloul et al.'s PBS solver on most examples.

4 A Scheduler with a Web Interface

Based on the techniques presented in the previous sections, we build a scheduler available to the public on the Internet for creating fair game schedules of a tournament. The web address of the scheduler is

`http://ar.cs.uiowa.edu/cgi-bin/schedule.cgi`

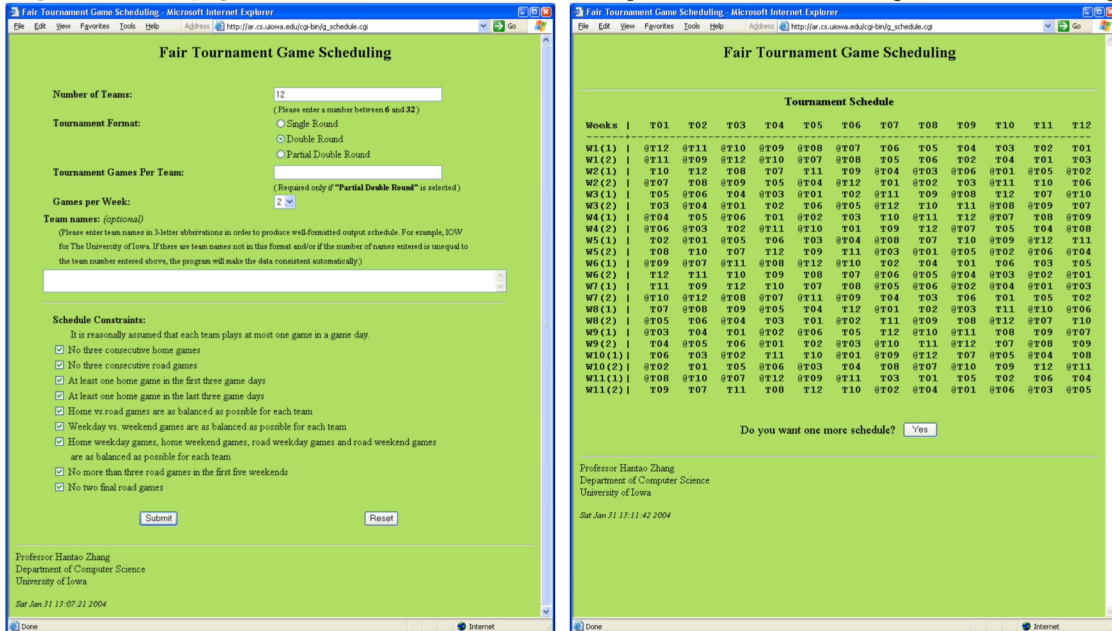
and the web page is shown in Fig. 1(a).

The first parameter from the user is the number of teams n. The user then decides how many games played by each team by choosing either a single round-robin tournament ($n - 1$ games); or a double round-robin tournament ($2(n - 1)$ games); or a partial double round-robin tournament (m games, $n \leq m < 2(n - 1)$).

Besides that each team plays about the same number of road games and home games, there are also other fairness criteria. For a schedule to be fair, we may require that no teams play three consecutive home games or three consecutive road games. If we take a look at the actual BigTen regular season schedule [16], we can see that the schedule is far from fair.

The following fair constraints are offered by the scheduler:

Fig. 1. The web page of the tournament scheduler at `http://ar.cs.uiowa.edu/cgi-bin/schedule.cgi`



(a) Input Form **(b)** Output Form

- No three consecutive home games.
- No three consecutive road games.
- At least one home game in the first three game days.
- At least one home game in the last three game days.
- Home vs. road games are as balanced as possible for each team.
- Weekday vs. weekend games are as balanced as possible for each team.
- Home weekday games, home weekend games, road weekday games and road weekend games are as balanced as possible for each team.
- No more than three road games in the first five weekends.
- No two final road games.

Note that all the above constraints can be easily specified using the boolean variables $p_{x,y,z}$ introduced in Section 2. After the user selects the desired constraints, the scheduler does the following three tasks.

1. Send the selected information to a program which generates corresponding ordinary and TL clauses in the extended DIMACS CNF format.
2. Call the SAT solver on the generated clause set to search for models.
3. Once a SAT model is found, another program is called to interpret the SAT model into a scheduling table.

If the user wants to see more schedules, the steps 2-3 above are repeated.

An example of the schedule produced by the scheduler is illustrated in Fig. 1 (b), where the number of teams is 12 for a double round-robin tournament, with all the constraints selected. The scheduler took only a few seconds for this schedule.

5 Conclusion

We have presented a scheduler with a web interface for generating fair game schedules of a tournament. The tournament can be either single or double round-robin or something in between. The search engine inside the scheduler is a SAT solver which can handle a mix of ordinary and TL clauses. The latter are the formulas using the function TL which counts the number of true literals

in a clause. By using TL clauses, we could solve the scheduling problems in a few of seconds. If we convert them into ordinary clauses, the state-of-the-art SAT solvers could not solve them in one week. We showed how to integrate TL clauses into a SAT solver and take advantages of the advanced SAT techniques. We hope that our scheduler can provide a useful service to the people who are interested in fair sports schedules.

Some of the techniques for handling TL-clauses, such as the treatment of *ancestors* and constant cost for backtracking, can be extended to handle linear constraints $\sum_{i=1}^{n} a_i x_i \leq b$, where a_i, b are positive integers and x_i are 0-1 variables. In general, it is more expensive to handle linear constraints than TL clauses. For instance, we may have to scan each literal in the constraint whenever one of the x_i is assigned 1; for TL clauses, the literals are scanned only when the TL-counter is less than 1. Sorting the literals according to a_i may help to reduce this cost. Further research is needed to confirm these statements.

References

1. Aloul, F., Ramnai, A., Markov, I., and Sakallah, Generic ILP vs. specialized 0-1 ILP, an update. *ICCAD*, 450-457, 2002.
2. Audemard, G., Bertoli, P., Cimatti, A., Kornilowicz, A., Sebastiani, R., A SAT based approach for solving formulas over boolean and linear mathematical propositions, Proc. of CADE'02.
3. Cain, W.O., Jr.: The computer-assisted heuristic approach used to schedule the major league baseball clubs. In Ladany and Machol (eds.) *Optimal Strategies in Sports*, 5, Studies in Management Science and Systems, pp. 32-41. North-Holland Pub., 1977.
4. Crawford, J.M, Baker, A.B., Experimental results on the application of satisfiability algorithms to scheduling problems. In Proceedings of the 12th AAAI, 1092–1097, 1994. `http://citeseer.nj.nec.com/crawford94experimental.html`
5. (1995) Freeman, J.W., Improvements to propositional satisfiability search algorithms. Ph.D. Dissertation, Dept. of Computer Science, University of Pennsylvania.
6. Hamiez, J., Hao, J., Solving the sports league scheduling problem with tabu search, Lecture Notes in Computer Science, 2148, pp24–36, 2001.
7. Henz, M.: Scheduling a major college basketball conference – revisited. *Operation Research*, 49(1), 2001.
8. Henz, M.: Friar Tuck 1.1: A constraint-based round robin planner. The software is available at `http://www.comp.nus.edu.sg/~henz/projects/FriarTuck`, 2002
9. Marriott, K., Stuckey, P.: Programming with constraints. MIT Press, Cambridge, MA, 1998.
10. Nemhauser, G., Trick, M.: Scheduling a major college basketball conference. *Operation Research*, 46(1), 1998.
11. Schaerf, A., Scheduling sport tournaments using constraint logic programming, In W. Wahlster (ed): Proceedings of the 12th European Conference on Artificial Intelligence (ECAI-96), Wiley & Sons, pp634–638, 1996.
12. Schruder J.A.M.: Combinatorial aspects of construction of competition dutch professional football leagues. *Discrete Applied Mathematics*, 35, 301-312, 1992.
13. Silva, J.P.M., Sakallah, K.A., Conflict analysis in search algorithms for propositional satisfiability. Technical Reports, Cadence European Laboratories, ALGOS, INESC, Lisboa, Portugal, May 1996.
14. Wallace, M.: Practical applications of constraint programming. Constraints, 1(1&2), 139-168.
15. de Werra, D.: Some models of graphs for scheduling sports competitions. *Discrete Applied Mathematics*, 21, 47-65, 1988.
16. Yahoo!Sports `http://sports.yahoo.com/ncaab/standings.html`, 2004.
17. Zhang, H.: Specifying Latin squares in propositional logic, in R. Veroff (ed.): Automated Reasoning and Its Applications, Essays in honor of Larry Wos, Chapter 6, MIT Press, 1997.
18. Zhang, H.: Sato: An efficient propositional prover, Proc. of International Conference on Automated Deduction (CADE-97). pp. 308–312, Lecture Notes in Artificial Intelligence 1104, Springer-Verlag, 1997.
19. Zhang, H., Generating college conference basketball schedules by a SAT solver, Fifth International Symposium on the Theory and Applications of Satisfiability Testing, 2002.
20. Zhang, H., Efficient data structure for clauses in SAT solvers, *Intl. Workshop on Microprocessor Testing and Verification* (MTV03), Austin, TX.
21. Zhang, H., Stickel, M.: Implementing the Davis-Putnam method, *J. of Automated Reasoning* 24: 277-296, 2000.
22. Zhang, L., Madigan, C., Moskewicz, M., Malik, S., Efficient conflict driven learning in a boolean satisfiability solver, International Conference on Computer Aided Design (ICCAD), 2001.