Playing with Constraint Programming and Large Neighborhood Search for Traveling Tournaments

Martin Henz

National University of Singapore
henz@comp.nus.edu.sg

Abstract. Constraint programming can be used to solve small tournament scheduling problems to optimality. Beyonds its low size limits, local search techniques have been shown to yield close-to-optimal schedules, when augmented with simulated annealing, reheating, strategic oscillation and other techniques. In these approaches, the local moves are relatively small, making the moves fast, but requiring sophisticated mechanisms to escape local minima. This paper explores the possibility of making use of constraint programming as a technique for achieving large moves in local search. The proposed technique falls within the algorithm scheme known as very large scale neighborhood search.

1 Introduction

Local search [1] has been applied successfully to large scale optimization problems. Its applications include a wide variety of timetabling problems, including highly structured problems. The only disadvantage, its inability to prove the optimality of a solution, vanishes as the problem size grows and complete search techniques fail to deliver optimal solutions. The main technical problem of local search is to escape local minima in the search. The techniques proposed to alleviate the problem include simulated annealing, where uphill moves are performed with decreasing probability, and tabu lists, where recent moves are stored and avoided.

Very large scale neighborhood search [2] (VLNS) addresses the problem of local minima by defining very large neighborhoods, and exploring these neighborhoods with efficient algorithms to constitute single move during the search. Often, the neighborhoods chosen for VLNS are amenable to algorithms with low polynomial complexity, exploiting techniques such as dynamic programming.

In this paper, we propose to employ constraint programming as a mechanism for making local moves. The idea is that the constraints in a given schedule are relaxed by keeping only certain variable assignments and relaxing all other variables. A step in the local search then constitutes a branch-and-bound search for the optimal assignment of the relaxed variables. The motivation is to make use of constraint programming in order to explore very large neighborhoods, hoping that the resulting search escapes from local minima more easily compared to local search variants with smaller neighborhoods.

This technique might work well for tightly constrained problems, where constraint programming is known to be able to efficiently solve small instances. One such an application is the scheduling of round robin tournaments [6]. The traveling tournament problem [5, 13] (TTP) is a set of benchmark problems that combines round robin tournaments with a cost function that is affected by every slot of the schedule. This problem provides an ideal testing ground for the proposed search technique. The competition from local search is strong. The work in [3, 11] shows that local search can come within a few percent of proven lower bounds for even large problems. We currently cannot reach the quality of solutions achieved by local search with our approach, but nevertheless think the proposal justifies serious consideration.

The following section describes the TTP in detail. Section 3 describes the technique of constraint programming to prepare the reader for a constraint programming model of the TTP in Section 4. Section 5 introduces VLNS in more detail. Section 6 describes different ways to define neighborhoods for TTP in a VLNS setting. Section 7 proposes an algorithm for solving TTP with VLNS using constraint programming for local moves, and finally Section 8 gives preliminary experimental results.

2 The Traveling Tournament Problem

The TTP [5] asks for an optimal intermural double round robin tournament schedule for an even number of teams n. Every team i plays against every other team exactly twice during the competition, once at the place of team i (a home match for i) and once at the other team's place (an away match for i). The first of the two matches is called the first leg, the second is the return match. In each round of the tournament, every team plays exactly once. Therefore, the $n(n-1)$ matches must be distributed over $2n-2$ rounds. A team may have at most three home games in consecutive rounds and at most three away games in consecutive rounds. Return matches cannot be in the round immeditately following the corresponding first legs. The aim of the traveling tournament problem is to generate a schedule such that the overall travel cost incurred by the teams is minimized. An instance of the traveling tournament problem consists of an even number n of teams, and a symmetric $n \times n$ distance matrix d, such that each integer value $d_{i,j}$ represents the travel distance between the home stadiums of teams i and j. The travel distance for a team throughout the tournament is the distance to be traveled from its home to the first venue (if the first match is an away match), then on to the second venue, etc, to the last venue, and finally back home again (if the last match was an away match). The overall travel distance to be minimized is simply the sum of the travel distances of all teams.

To get the constraints even tighter (which we deem beneficial for the success of our approach), we focus on the mirrored variant of the problem, as described in [13, 11]. In this version, the return leg occurs always $n-1$ rounds after the corresponding first match, which means that the schedule consists of a single round robin followed by the same single round robin with inverted venues.

The traveling tournament problem attracted the attention of timetabling researchers, because it captures the essence of an important class of sports scheduling problems, and because it represents a highly structured timetabling problem with a large search space, and with a cost function that is directly affected by every slot in the timetable.

3 Constraint Programming

Finite-domain constraint programming is a technique designed for solving combinatorial search problems. Stuckey and Marriott [8] explain the approach in detail and Wallace [15] presents an overview of applications of finite-domain constraint programming.

Every variable of a model is represented by a finite-domain variable. A constraint store stores information on such a variable in the form of the set of possible values that the variable can take; this set is called the current domain of the variable. Some constraints can be directly entered in the constraint store. For example, the constraint $x \neq 5$ can be expressed in the constraint store by removing 5 from the domain of x. More complex constraints are translated by the programmer into computational agents called propagators. Each propagator observes the variables given by the corresponding constraint in the problem. Whenever possible, it strengthens the constraint store with respect to these variables by excluding values from their domain according to the corresponding constraint. The process of propagation continues until no propagator can further strengthen the constraint store. At this point, many problem variables typically have still non-singleton domains. Thus the constraint store does not represent a solution, and search becomes necessary.

Search for solutions is implemented by choice points. A choice point generates a constraint c. From the current stable constraint store cs, two new constraint stores are created by adding c and $\neg c$, respectively, to cs. In each branch, propagation leads to further domain reductions and possibly to empty domains of some variables, in which case the branch can be pruned.

The choice of the constraint c at each branching determines the shape and size of the search tree. A mechanism to systematically generate these constraints is called a search strategy. A common search strategy is called variable enumeration, where the constraints c have the form $x = v$ for some variable x and some value v from its domain.

In order to optimize solutions with respect to a cost function, the usual depth-first search is modified by a bounding constraint. To this aim, a bound variable *cost* is constrained such that it reflects the cost of the solution. Whenever a new solution is found with a cost C the constraint $cost < C$ is added to every store to be considered afterwards. We shall denote this branch-and-bound algorithm by an operator *branchandbound* which is applied to a given model with a distinguished cost variable and returns the optimal solution, and the usual constraint-based depth-first search algorithm without optimization by *depthfirst* with the same signature.

A constraint programming system takes care of activating propagators, reaching stability and performing the branch-and-bound search. The programmer can concentrate on translating the constraints into appropriate propagators and specifying the search strategy. Finite-domain constraint programming systems support this task through libraries of propagators and search strategies.

4 Constraint Programming Model for the Traveling Tournament Problem

Our constraint programming model *ttp* is defined as follows. For every team i and every round j of the tournament, a finite domain variable $op_{i,j}$ represents the opponent against which team i plays in round j. Each variable $op_{i,j}$ has the initial domain of $\{1, \ldots, i-1, i+1, \ldots, n\}$, allowing team i to play any other team in round j. To determine the venues, for every every team i and round j, a 0/1 variable $home_{i,j}$ expresses whether team i plays home in round j, and a 0/1 variable $away_{i,j}$ expresses whether team i plays away in round j.

The double round robin constraints are expressed with straightforward propagators. For some constraints, there is a choice between propagators of different strength.

- The symmetric all-different constraint that forces proper team matchings in a round is implemented by an all-different constraint and inequalities, instead of the constraint given in [10], since in practice the additional pruning does not justify the computational effort needed; see [7] for a more detailed analysis.
- The constraint that forces a team to play all other teams twice is implemented by a cardinality constraint [14], instead of a more complex scheme that would allow to use the all-different constraint, again as a result of a performance analysis.

The cost function is implemented by constraints that collaborate to restrict the variable *totaldistance*. This variable is used as bound variable in constraint-based branch-and-bound. To compute *totaldistance*, the travel distance for team i between round $j-1$ and round j is represented by a matrix $distance_{i,j}$, $1 \leq i \leq n$, $0 \leq j \leq 2n-2$, of finite domain variables.

Each variable $distance_{i,0}$ represents the distance traveled for a team i to reach its first destination. To constrain this variable, we first access the distance vector d_i via an element constraints [4] to obtain a preliminary distance $distance'_{i,0}$ as follows.

$$element(op_{i,0}, d_i, distance'_{i,0})$$

The final distance $distance_{i,0}$ is a result of multiplying the appropriate away variable with the preliminary distance variable.

$$times(away_{i,1}, distance'_{i,0}, distance_{i,0})$$

The variable $distance_{i,2n-2}$, representing the distance traveled for a team i from the venue of its last match back home, is constrained in a similar way.

The computation of the internal travel distance $distance_{i,j}$ for team i between rounds $j-1$ and j is more complex, but similar in style. There are four cases, distinguished by the corresponding venue variables $home_{i,j-1}$, $home_{i,j}$, $away_{i,j-1}$, and $away_{i,j}$. If both $home_{i,j-1}$ and $home_{i,j}$ are 1, we can set $distance_{i,j}$ to 0. The case where one of $home_{i,j-1}$ and $home_{i,j}$ is 0 and the other one is 1 is similar to the situation at the beginning and end of the schedule, and implemented using an element constraint involving either $op_{i,j}$ or $op_{i,j-1}$. In the case where both *home* variables are 0, we need to access the distance matrix using a two-dimensional variant of the element constraint, using both opponent variables $op_{i,j-1}$ and $op_{i,j}$ as indices.

Finally, all variables $distance_{i,j}$ are summed to the bound variable *totaldistance*.

As branching technique, we proceed team by team in random order. We first enumerate all *home* variables for the current team, and then all *op* variables, again in random order.

In our experiments, we can solve the smallest two benchmark problems $n = 4, 6$ using constraint programming alone (see Section 8).

5 Very Large Scale Neighborhood Search

Local search starts with an initial schedule and repeatedly applies local moves to it. Local moves are typically small changes to the schedule that improve the overall cost function. By moving to better and better schedules, local search quickly finds local minima. Much attention is devoted to the question how to escape local minima in search for more global minima. Many different factors determine the success of local search, including the chosen model and the escape mechanism. The factor that we are focussing on here is the size of the neighborhood from which the next move is chosen. Ahuja et al [2] describe the issue as follows:

> A critical issue in the design of a neighborhood search approach is the choice of the neighborhood structure, that is the manner in which the neighborhood is defined. This choice largely determines whether the neighborhood search will develop solutions that are highly accurate or whether they will develop solutions with very poor local optima. As a rule of thumb, the larger the neighborhood, the beter is the quality of the locally optimal solutions, and the greater is the accuracy of the final solution that is obtained. At the same time, the larger the neighborhood, the longer it takes to search the neighborhood at each iteration. (R.K. Ahuja, O. Ergun, J.B. Orlin, A.P. Punnen ([2])

The focus of very large scale neighborhood search is to define neighborhoods large enough to easily escape local minima while still being able to search the neighborhood for an optimal move. Ahuja et al [2] categorize the algorithms for exploring the neighborhoods into three classes:

- Heuristic search methods that explore exponentially large neighborhoods, defined by a suitably chosen "depth" (neighborhood size),

– Network flow-based improvement algorithms, and
– Polynomial algorithms resulting from suitably restricted NP-hard problems.

In this paper, we propose to define the neighborhood of a given schedule by keeping the values of certain variables as constraints. Since constraint programming has been shown to be a suitable technique for efficiently solving small tournament problems, we choose this technology to explore the resulting neighborhoods for optimal schedules.

6 Defining Neighborhoods

In large neighborhood search, a neighborhood is defined by relaxing the constraints given by a candidate schedule. We are exploring different techniques for relaxing a given schedule, denoted by an operator *relax*, which is applied to a given model and a solution and returns a model in which the variables that are not relaxed are fixed as in the given solution.

Relaxing rounds. In this relaxation technique, all variables of a set of rounds are relaxed. We can often relax the majority of rounds, leaving only the opponents and venues of a few rounds fixed as given in the current schedule. This relaxation allows the optimization to shuffle entire rounds of the schedule as in the *SwapRounds* move operation in [3], as well as rearrangements of opponents and venues within the relaxed rounds.

Relaxing teams. Here, all variables of a set of teams are relaxed. Again, we can often relax the majority of teams, leaving only the opponents and venues of a few teams fixed. This relaxation allows the optimization to shuffle the schedule of entire teams as in the *SwapTeams* move operation in [3], as well as rearrangements of opponents and venues within the relaxed teams.

Random Relaxation. In this relaxation technique, we randomly choose team/round slot coordinates and relax all opponent and venue variables that belong to the chosen slots.

Relaxing Opponents. Here, all opponent variables $op_{i,j}$ are relaxed, which means that the home/away patterns are kept for the teams, and the opponents are rearranged. If that does not lead to a sufficiently small neighborhood, we can combine this relaxation technique with any of the previous, for example random relaxation.

Relaxing Venues. Here, all home variables $home_{i,j}$ and away variables $away_{i,j}$ are relaxed, which means that the opponents are kept for the teams, and the venues are rearranged. If that does not lead to a sufficiently small neighborhood, we can combine this relaxation technique with any of the previous, for example random relaxation.

To get a good coverage of the search space during the search, it is useful to alternate between different relaxation techniques.

7 Putting it all together

The VLNS techniques we are proposing generates a first solution and then repeatedly relaxes the solution and computes the best solution within the neighborhood defined by the relaxation. Our VLNS algorthms are therefore instances of the following algorithm scheme.

```
$s = depthfirst(ttp)$
$while\ \neg(termination\ condition)$
    $s = branchandbound(relax(ttp, s))$
```

Escape mechanism. The search usually gets stuck in a local minimum regardless how large a neighborhood size is chose. We therefore regularly restart the search with a fresh initial solution.

Tabu list. We want to avoid that branch-and-bound finds the current solution as best solution. We therefore implement a simple tabu mechanism that excludes solutions with a cost that was encountered already before. Tabu tenure is indefinite in our implementation.

Time limit. The computing time for a branch-and-bound search in a neighborhood defined by relaxation is not predictable. In order to force the local search to progress at a predictable rate, we impose a time limit on each branch-and-bound search. There are four possible results of the branch-and-bound.

1. Branch-and-bound finds the optimal solution and terminates.
2. Branch-and-bound finds solutions but does not manage to prove the optimality of the last solution within the time limit.
3. Branch-and-bound finds no solution and terminates, which means that it proves that there is no solution. This is possible due to the tabu mechanism.
4. Branch-and-bound finds no solution, but does not manage to prove that there is no solution.

In the last two cases, we simply ignore the corresponding neighborhood search. In the first two cases, we adopt the last solution as the new current solution. In the second case, an alternative approach would be to accept the last solution only if it is not much worse than the previous solution.

Self-tuning neighborhood size. The size of the neighborhood defined by each of the relaxation techniques in Section 6 can be tuned by parameters. This makes the algorithm difficult to handle. Using the time limit approach just described, we can make these parameters self-tuning. If the neighborhood search exhausts the time limit without finding any solutions (case 4 above), the neighborhood is too large, and we reduce the corresponding relaxation parameter. On the other hand, if the neighborhood search proves that there are no solutions, the neighborhood is too small, and we increase the corresponding relaxation parameter.

Simulated annealing and strategic oscillation. Using some ideas from [3], we can employ simulated annealing. One possibility would be to increase the time limit over time, which will force the search to be more focussed as time goes by. The techniques of reheating and strategic oscillation in [3] can also be used in the context of VLNS.

8 Preliminary Results

The results obtained so far are very preliminary. They were obtained using an experimental implementation in the constraint programming language Oz [12] using the Mozart programming system [9] on a Pentium IV with 2.4GHz and 512MB running Windows XP.

For the problem sizes of 4 and 6 teams, constraint programming alone (without local search) can generate the optimal solution. For 4 teams, the computing time is a few microseconds and a few dozens of search nodes, whereas 6 teams takes more than 10 minutes and more than 100,000 nodes.

All VLNS experiments were done using the simplest constraint programming model using the basic algorithm, augmented by a rotating neighborhood selection scheme. We restart every 100 moves and use a time limit of 1 second (for the smallest problems) to 1 minute (for the largest problems). We alternate between opponent relaxation, venue relaxation and random relaxation, using self-tuning neighborhood size selection.

For 8 teams, we have obtained a solution with cost of 42142 (the best solution obtained in [11] is better at 41928). For 10 teams, we achieve a cost of 65464 (again, the best solution by Ribeiro is better at 64976), and for 12 teams 135950 (120696).

We have experimented with simulated annealing with strategic oscillation as described in the previous section, but have not achieved an improvement over the basic algorithm.

More work is needed to be able to compete with more conventional local search techniques. However, we are convinced that if constraint programming is to continue to contribute to tournament scheduling, it may be along the lines of very large neighborhood search.

Acknowledgements

Zhang Da contributed to this work with helpful discussions. This work was supported in part by FriarTuck Pte Ltd.

References

1. E.L. Aarts and J. K Lenstra. *Local Search in Combinatorial Optimization.* Princeton University Press, 1997.
2. R.K. Ahuja, O. Ergun, J.B. Orlin, and A.P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123:75–102, 2002.

3. A. Anagnostopoulos, L. Michel, P. Van Hentenryck, and Y. Vergados. A simulated annealing approach to the traveling tournament problem. In *Proceedings of the 5th International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR 2003)*, Montreal, Canada, May 2003.
4. Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In Yves Kodratoff, editor, *Proceedings of the European Conference on Artificial Intelligence*, pages 290–295, Munich, Germany, August 1988. Pitman Publishers, London.
5. K. Easton, G. Nemhauser, and M. Trick. The traveling tournament problem description and benchmarks. In Toby Walsh, editor, *Principles and Practice of Constraint Programming—CP 2001, Proceedings of the Seventh International Conference*, Lecture Notes in Computer Science 2239, pages 580–589, Cyprus, 2001. Springer-Verlag, Berlin.
6. Martin Henz. Scheduling a major college basketball conference—revisited. *Operations Research*, 49(1), January 2001.
7. Martin Henz, Tobias Müller, and Sven Thiel. Global constraints for round robin tournament scheduling. *European Journal for Operational Research*, 153(1):92–101, February 2004.
8. Kim Marriott and Peter J. Stuckey. *Programming with Constraints.* The MIT Press, Cambridge, MA, 1998.
9. Mozart Consortium. The Mozart Programming System. Documentation and system available from `http://www.mozart-oz.org`, Programming Systems Lab, Saarbrücken, Swedish Institute of Computer Science, Stockholm, and Université catholique de Louvain, 2004.
10. Jean-Charles Régin. The symmetric alldiff constraint. In Thomas Dean, editor, *Proceedings of the International Joint Conference on Artificial Intelligence*, volume 1, pages 420–425, Stockholm, Sweden, August 1999. Morgan Kaufmann Publishers, San Mateo, CA.
11. C.C. Ribeiro and S. Urrutia. Heuristics for the mirrored traveling tournament problem. In Edmund Burke and Michael Trick, editors, *Practice and Theory of Automated Timetabling, Fifth International Conference, PATAT 2004*, Lecture Notes in Computer Science, Pittsburgh, USA, 2004. Springer-Verlag, Berlin.
12. Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science 1000, pages 324–343. Springer-Verlag, Berlin, 1995.
13. Michael Trick. Challenge traveling tournament instances. http://mat.gsia.cmu.edu/TOURN/, 2004.
14. P. Van Hentenryck and Y. Deville. The cardinality operator: A new logical connective for constraint logic programming. Report CS-90-24, Brown University, November 1990.
15. Mark Wallace. Practical applications of constraint programming. *Constraints*, 1(1&2):139–168, 1996.