Solving the Travelling Umpire Problem with Answer Set Programming

Joost Vennekens(✉)

Department of Computer Science @ Technology Campus De Nayer, KU Leuven,
J.-P. De Nayerlaan 5, 2860 St-katelijne-waver, Belgium
joost.vennekens@cs.kuleuven.be

Abstract. In this paper, we develop an Answer Set Programming (ASP) solution to the Travelling Umpire Problem (TUP). We investigate a number of different ways to improve the computational performance of this solution and compare it to the current state-of-the-art. Our results demonstrate that the ASP solution is superior to other declarative solutions, such as Constraint Programming, but that it cannot match the most recent special-purpose algorithms from the literature. However, when compared to the earlier generation of special-purpose algorithms, it does quite well.

1 Introduction

The Travelling Tournament Problem is a well-known optimisation problem, where the goal is to schedule a series of games at different venues such that the travel distance for the participating teams is minimized. The *Travelling Umpire Problem (TUP)* is a related problem, in which the games and their location are given and the goal is to assign an umpire to each of these games. This assignment must satisfy a number of hard constraints (e.g., the same umpire should not referee the same team two weeks in a row) and the overall distance travelled by the umpires should be minimized.

This problem, which abstracts a real-life task for the Major League Baseball (MLB), was first introduced into the scientific literature in 2007 [15]. Over the last five years, it has received a good deal of attention within the scientific community [4,5,13,14,16–18,20,21]. One of its interesting characteristics is that the TUP has a compact description, but is computationally very challenging. Its associated decision problem is NP-complete [5].

In this paper, we investigate whether it is possible to solve the TUP using the declarative paradigm of Answer Set Programming (ASP). For this, we will make use of the state-of-the-art ASP solver Clasp [8], which is a regular winner of the ASP competition [1–3,6,9,10]. As we will show below, the TUP can be represented in ASP in an elegant and modular representation. Such a representation has the advantage that it can easily be extended with additional constraints, in order to better capture the real-life MLB problem. The computational performance of our method is superior to that of other declarative paradigms, and

T. Bosse and B. Bredeweg (Eds.): BNAIC 2016, CCIS 765, pp. 106–118, 2017.
DOI: 10.1007/978-3-319-67468-1_8

matches that of the first special-purpose algorithm to be developed for TUP [16]. However, it cannot match the performance of more recent special-purpose algorithms, such as the state-of-the-art method from [14]. Nevertheless, it is of course a benefit of our declarative approach that improvements to the general-purpose Clasp server automatically speed up our approach as well.

This paper is structured as follows. Section 2 recalls the basic concepts of ASP and Sect. 3 the definition of the TUP. In Sect. 4, our basic encoding of TUP in ASP is then presented. Next, Sect. 5 discusses some ways of improving the computational performance of this encoding. We evaluate the performance of our method and compare it to existing approaches in Sect. 6. Finally, Sect. 7 presents our conclusions.

2 Preliminaries: Answer Set Programming

In this section, we briefly recall the Answer Set Programming (ASP) language and it semantics [11].

An *atom* is an expression $p(T_1, \ldots, T_n)$ where p is an n-ary predicate symbol ($n \geq 0$) and the T_i are either constants (starting with a lower case letter) or variables (starting with an upper case letter). A *(normal) rule* is an expression of the form:

$$A \ :- \ B_1, \ldots, B_n, \texttt{not}\ C_1, \ldots, \texttt{not}\ C_m. \tag{1}$$

Here, A and all of the B_i and C_j are atoms. The atom A is called the *head* of the rule and $B_1, \ldots, B_n$, not $C_1, \ldots,$ not C_m its *body*. A *(normal) logic program* is a finite set of such rules. An atom, rule or program is called *ground* if it does not contain any variables.

A *belief set* X is a set of ground atoms. Such a set X *satisfies* a ground rule r of form (1) if A belongs to X or there exists an $i \in [1, n]$ such that $B_i \notin X$ or a $j \in [1, m]$ such that $C_j \in X$. A belief set is a model of a ground program P if it satisfies all rules $r \in P$.

For a ground rule r of the form (3) and a belief set X, the reduct r^X is defined whenever there is no atom C_j for $j \in [1, m]$ such that $C_j \in X$. If the reduct r^X is defined, then it is the rule: $A \ :- \ B_1, \ldots, B_n$. The reduct P^X of the ground program P consists of all r^X for $r \in P$ for which the reduct is defined. A belief set X is a *stable model* or *answer set* of P, denoted $X \models_{st} P$, if it is the least model of P^X.

The answer sets of a non-ground program are defined as the answer sets of the *grounding* of the program. This grounding is constructed by replacing each non-ground rule r by the set of all ground rules that can be constructed by replacing all the variables in r by constants in all possible ways.

An answer set solver is a program that computes the stable models of a given program. Typically, these solvers extend the basic ASP language with some additional constructs to make it easier to represent interesting problems. In this paper, we will use the input language of the solver Clasp [7]. This language allows the following additional constructs.

A rule of the form

$$:- B_1, \ldots, B_n, \texttt{not}\ C_1, \ldots, \texttt{not}\ C_m, \tag{2}$$

is called a *constraint*. It is seen as an abbreviation for a rule

$$f\ :- B_1, \ldots, B_n, \texttt{not}\ C_1, \ldots, \texttt{not}\ C_m, \texttt{not}\ f. \tag{3}$$

where f is a fresh atom. The effect of such a rule is that no answer set may contain both all of the positive literals B_i and none of the negative literals C_i. A constraint therefore expresses that its body may *not* be satisfied.

Another common pattern in ASP programming is the use of a loop over negation to represent a choice between two alternatives. For instance, to express that either p or q must hold, the following two rules may be used:

```
p :- not q.     q :- not p.
```

The answer sets of this small program are precisely {p} and {q}. However, because this style of programming quickly grows cumbersome, Claps also allows to represent choices as:

```
1 { p, q} 1.
```

This expresses that precisely one of the two atoms p and q must holds (the leftmost 1 states that at least one of these atoms must hold and the rightmost 1 states that at most one may hold). It has the same two answer sets as the above two rules (but the two programs are not strongly equivalent—for that, an additional constraint `:- p,q` should be added to the two rules).

This construct is made even more powerful by the fact that more complex set expressions may be used instead of a simple enumeration of atoms. For instance, `1{p(X) : q(X)}1` states that precisely one atom $p(X)$ must hold out of all atoms $p(X)$ for which X is such that $q(X)$ holds. If we add to this rule the following three facts:

```
q(1).   q(2).  q(3).
```

the answer sets will be precisely {p(1)}, {p(2)} and {p(3)}. The above three facts may also be conveniently abbreviated as: `q(1..3).`

3 The Travelling Umpire Problem

The TUP is the problem of assigning n umpires to a double round-robin tournament of $2n$ teams. Since each team plays every other team twice, there are $r = 2(2n - 1)$ different rounds, in which each team plays against exactly one other team. Each game is played at the home venue of one of the two participating teams. The assignment of umpires is subject to three hard constraints:

- Each umpire should visit each team's home venue at least once.
- An umpire should not visit the same home venue more than once in any sequence of q_1 rounds.
- An umpire should not referee the same team more than once in any sequence of q_2 rounds.

The parameters q_1 and q_2 control the tightness of the constraints: higher values are more difficult. Clearly, it only makes sense to have $q_2 \leq q_1 \leq r/2$.

In addition to these three hard constraints, there is also an objective function that must be minimized, namely, the total distance travelled by all of the umpires (assuming they start in the location where their first game is played). To calculate this objective function, a (symmetric) distance matrix is given which enumerates the distances between the home venues of each pair of teams.

Figure 1 shows an example instance for $n = 2$ and a solution for the problem with $q_1 = 2$ and $q_2 = 1$ (note that this problem has no solutions for $q_2 > 1$; taking $q_2 = 1$ means that the third constraint can simply be ignored). The solution shown is also the optimal solution (regardless of which distance matrix is used), since the only other solution that exists (modulo the symmetry between the two umpires) is that in which they needlessly move around between rounds 3 and 4.

Round	Home	Away	Umpire
0	Team 1	Team 2	?
	Team 3	Team 4	?
1	Team 1	Team 3	?
	Team 4	Team 2	?
2	Team 2	Team 3	?
	Team 4	Team 1	?
3	Team 2	Team 1	?
	Team 4	Team 3	?
4	Team 3	Team 1	?
	Team 2	Team 4	?
5	Team 3	Team 2	?
	Team 1	Team 4	?

→

Round	Home	Away	Umpire
0	Team 1	Team 2	Umpire 1
	Team 3	Team 4	Umpire 2
1	Team 1	Team 3	Umpire 2
	Team 4	Team 2	Umpire 1
2	Team 2	Team 3	Umpire 1
	Team 4	Team 1	Umpire 2
3	Team 2	Team 1	Umpire 2
	Team 4	Team 3	Umpire 1
4	Team 3	Team 1	Umpire 2
	Team 2	Team 4	Umpire 1
5	Team 3	Team 2	Umpire 1
	Team 1	Team 4	Umpire 2

Fig. 1. An instance of the TUP for $n = 2, q_1 = 2, q_2 = 1$ and its optimal solution.

4 Representing the TUP in ASP

Problem instance. The parameters of an instance are represented by a set of ASP facts

`limit_big`(q_1). `limit_small`(q_2).
`team`(1..$2n$). `umpire`(1..n). `round`(0..r).

The tournament schedule is given by a set of facts of the form `plays`(i, j, r), which denote that home-team i plays away-team j in round r. The distance matrix is represented by a set of facts `distance`(i, j, δ), indicating that the distance between the home venue of team i and that of team j is δ. Only facts for $i \leq j$ are included. The symmetric information is represented by a rule

```
dist(X,Y,D) :- dist(Y,X,D).
```

Finally, it is also convenient to include a unary predicate `last`$(r-1)$ which indicates the number $r-1$ of the last round and a unary predicate `max_dist`(δ_{max}) which indicates the largest number that occurs in the distance matrix.

The example shown in Fig. 1 corresponds to the following set of facts:

```
limit_big(2).    plays(2,3,2).  dist(1,2,745).  dist(3,3,0).
limit_small(1).  plays(4,1,2).  dist(1,3,665).  dist(3,4,380).
team(1..4).      plays(2,1,3).  dist(1,4,929).  dist(4,1,929).
umpire(1..2).    plays(4,3,3).  dist(2,1,745).  dist(4,2,337).
round(0..5).     plays(3,1,4).  dist(2,2,0).    dist(4,3,380).
plays(1,2,0).    plays(2,4,4).  dist(2,3,80).   dist(4,4,0).
plays(3,4,0).    plays(3,2,5).  dist(2,4,337).  max_dist(929).
plays(1,3,1).    plays(1,4,5).  dist(3,1,665).  last(5).
plays(4,2,1).    dist(1,1,0).   dist(3,2,80).
```

Generating the search space. We represent the possible solutions to the problem by a set of atoms `move`(u, t, r), meaning that umpire u moves to the home venue of team t in round r. For instance, `move(1,1,0)` means that umpire 1 moves to the home venue of team 1 in the round 0 (i.e., the home venue of team 1 is the starting position of umpire 1). The solution shown in Fig. 1 would be represented as:

```
move(1,1,0).    move(1,4,3).  move(2,3,0).    move(2,2,3).
move(1,4,1).    move(1,2,4).  move(2,1,1).    move(2,3,4).
move(1,2,2).    move(1,3,5).  move(2,4,2).    move(2,1,5).
```

Obviously, in a valid solution these atoms must be such that team t actually plays a home game in round r. We first define which teams this are.

```
home_team(Home,R) :- plays(Home,Away,R).
```

We can then generate the search space by the following choice rule:

```
1 { move(X,Y,T) : home_team(Y,T) } 1 :- umpire(X), round(T).
```

Testing the hard constraints. Each game must be assigned precisely one umpire. We represent this requirement as:

```
% Each game is officiated by at most one umpire
:- move(U1,T,R), move(U2,T,R), U1 != U2.

% Each game is officiated by at least one umpire
:- home_team(T), round(R), { move(U,T,R) : umpire(U) } 0.
```

The expression `{move(U,T,R) : umpire(U)} 0` holds if the set of all `move(U,T,R)` atoms for which `round(R)` holds is at most zero; in other words, if no umpires are assigned to the round R game in which team T is the home team. Given the way in which the search space is generated, it would actually be sufficient to add only of these two constraints. However, adding both of them helps the solver to solve the problem more quickly.

Next, we handle the constraint that each umpire should visit each venue. First, we define when an umpire U visits the home venue of team T:

```
been_to(U,T) :- round(R), move(U,T,R).
```

Now, the constraint is that this predicate must hold for all U and T:

```
:- umpire(U), team(T), not been_to(U,T).
```

Of the two constraints regarding repeated games with the same umpire, the constraint concerning home venues is easiest to represent (recall that `limit_big` contains the paramater q_1 of the instance):

```
:- move(U,T,R1), move(U,T,R2), R1 < R2, limit_big(B),
   R2 - R1 + 1 <= B.
```

To express the constraint regarding q_2, we first need to define when an umpire U officiates a game in which team T is involved (as either home or away team) in round R:

```
officiates(U,Home,R) :- move(U,Home,R).
officiates(U,Away,R) :- move(U,Home,R), plays(Home,Away,R).
```

Using this predicate, the constraint for the parameter q_1 (given by the predicate `limit_small`) is represented as follows:

```
 :- officiates(U,T,R1), officiates(U,T,R2),
    R1 < R2, limit_small(S), R2 - R1 + 1 <= S.
```

Optimisation. The rules and constraints that we have so far suffice to produce feasible solutions. To find the optimal solution, we first define the distance D that a given umpire U travelled in round R (i.e., to go from his venue in round $R-1$ to his venue for round R):

```
moved(U,R,D) :- umpire(U), team(T), round(R), R > 0,
                move(U,T,R), move(U,Tp,R-1), dist(T,Tp,D).
```

The following statement instructs Clasp to minimize the sum of all these distances D:

```
#minimize { D,U,R : moved(U,R,D) }.
```

5 Improving the Performance

The program that we have discussed so far correctly generates optimal solutions to the TUP. In this section, we discuss two additions to the program that may improve its computational performance.

A typical method for improving the efficiency of such a program is to introduce additional constraints to break symmetries. For the TUP, only one symmetry is known in the literature, namely the fact that the umpires are all identical. We therefore introduce the following symmetry breaking rule, which forces one particular assignment in round 0:

```
:- move(U1,T1,0), move(U2,T2,0), U1 < U2, T1 >= T2.
```

A second possibility for improving the efficiency is to choose a suitable heuristic to guide the search process. Similar to, e.g., SAT solvers, an ASP solver works by iteratively selecting some atom that is currently neither true nor false, assigning it one of these two truth values, and then propagating the effects of this assignment. A heuristic is used to decide, first of all, which atom (among those that do not yet have a truth value) should be selected and, second, whether this atom should be made true or whether it should be made false. Clasp already contains a number of built-in heuristics, but it also allows users to provide a domain-specific heuristic by defining a predicate `_heuristic` (a, x, w). Here, a is an atom, w is a weight and x is a special constant. If $x =$ `sign`, then w must be either 1 or -1. The meaning of such an atom `_heuristic(`a `,sign,-1)` is that whenever atom a is chosen, it will be given the value false; if instead $w = 1$, then a is assigned the value true.

For the TUP, a good heuristic might be to make choices that assign an umpire to a particular game (i.e., that make a `move(U,T,R)` atom true). It seems likely that these are the kind of choices that will lead to the most propagation: indeed, making a single such atom true will have at least the effect of forcing all $n - 1$ such atoms for the same game to be false. We therefore include the following rule:

```
_heuristic(move(U,T,R),sign, 1) :- round(R), team(T), umpire(U).
```

To complete the heuristic, it still remains to decide *which* `move(U,T,R)` atom to choose first. To get the most propagation, it seems reasonable to handle the different rounds in order, and since the assignment in the first round is fixed by the symmetry breaking rule, it makes sense to work in ascending order. Within a round, we greedily attempt to send each umpire to the venue with the smallest travel distance. In this way, we hope to bias the search towards more optimal solutions. This is the same strategy as followed in [14]. We implement this strategy by assigning each `move(U,T,R)` atom a weight of $n\delta_{max} + (\delta_{max} - \delta)$, where δ_{max} is the maximum distance between venues, δ is the distance that the umpire would have to travel to get to the home venue of team T in round R, and n is the number of rounds left after the Rth round.

```
_heuristic(move(U,T,R),level,(L-R+1) * M + (M-D) ) :-
      last(L),round(R),team(T),umpire(U),
      move(U,T1,R-1),dist(T,T1,D),max_dist(M).
```

In addition to changing the heuristic, Clasp also has numerous configuration options that determine how it precisely conducts the search for a solution. Because the space of all possible configurations is huge, a tool called Piclasp has been developed, which tries to automatically deduce an optimal configuration for a given problem using the SMAC method [12]. It does this by first experimenting with different settings on a number of training instances and then applying Machine Learning techniques to surmise an optimal configuration. This tool does not yet offer support for optimisation problems, however. Nevertheless, we have used it to determine a configuration for the problem of finding a feasible solution (i.e., one that satisfies all the hard constraints, without taking the optimisation criterium into account).

Finally, Clasp is also able to take advantage of multiple processors by running certain computations in parallel. It offers two strategies for this. The *compete* strategy launches a number of independent executions of the solver, each with different configuration parameters. The output is then simply the result produced by the best configuration. The *split* strategy essentially runs a single instance of the solver, splitting its search tree over different parallel threads. This has the advantage that the different threads actually help each other, rather than competing with each other and possible duplicating a lot of the work done by the other threads. The disadvantage, however, is that the different threads now need to communicate with—and thus wait for—each other, which may cause some of the capacity to go unused.

6 Experimental Results

In this section, we present some experiments that were conducted to determine which ways of running Clasp perform best for the TUP and how our solution compares to the state of the art. For the first point, we want to answer three questions:

- Whether the domain-specific heuristic discussed in the previous section performs better than Clasp's default heuristic (which it selects based on some properties of the problem instance);
- Whether the configuration settings learned by Piclasp perform better than the default configuration settings (which Clasp again selects based on properties of the problem instance);
- Whether using multiple processors in parallel leads to better results and, if so, which of the two possible strategies (*compete* or *split*) is better.

Experimental results are shown in Table 1. In the name of the different variants, `def` and `dom` refer to the use of, respectively, Clasp's default heurstic versus our domain-specific heuristic; `1`, `comp` and `split` refer to whether only a single

processor was used, or whether all available processors were used with either the *competitive* or *split* strategy; finally, π is added to the name whenever the settings learned by Piclasp where used instead of the default settings. All results in this table were computed on a Linux machine containing eight Intel(R) Core(TM) 3.2 GHz processors. The benchmarks instances come from [16] and a timeout of 10 min was used. All results in this section are given as the fraction difference with the best known solution from [19]. In other words, if x is the best value that was computed at the time-out and b is the best known value, then the tables report $\frac{x-b}{b}$. In addition, the table also reports the average fraction difference for each of the variants, the number of benchmarks for which the variant failed to produce a feasible solution with the time limit (indicated as `NoSol` in the table—these values are not taken into account when computing the average percentage difference), and the number of benchmarks won by this variant.

Table 1. Comparison of different variants for size $n = 14$.

Benchmark	def-comp	dom-split-π	def-1-π	def-1	dom-split	dom-1	dom-comp	dom-1-π	def-split	dom-comp-π
14-5-3	0.14	0.18	0.20	0.15	0.13	0.15	**0.13**	0.15	0.14	0.17
14-6-3	0.10	0.13	0.19	0.12	0.11	0.11	0.11	0.16	**0.09**	0.13
14-7-3	0.07	0.10	0.14	0.06	0.07	0.07	0.07	0.13	**0.06**	0.07
14A-5-3	0.17	0.21	0.23	0.18	**0.16**	**0.16**	0.16	0.21	0.16	0.19
14A-6-3	0.10	0.14	0.23	0.12	0.12	**0.10**	0.13	0.18	0.11	0.15
14A-7-3	**0.06**	0.12	0.20	0.11	0.09	0.09	0.07	0.11	0.08	0.12
14B-5-3	0.14	0.17	0.18	0.16	**0.13**	0.15	0.14	0.18	0.14	0.17
14B-6-3	0.12	0.14	0.15	0.13	**0.11**	0.12	0.12	0.14	0.13	0.16
14B-7-3	**0.07**	0.10	0.16	0.10	0.09	0.09	0.08	0.11	0.09	0.11
14C-5-3	0.16	0.18	0.22	0.18	**0.15**	0.17	0.16	0.21	0.19	0.18
14C-6-3	0.15	0.16	0.21	0.15	**0.10**	0.13	0.13	0.13	0.13	0.18
14C-7-3	**0.11**	0.13	NoSol	0.12	0.11	0.12	0.11	0.16	0.11	0.16
Avg	0.12	0.15	0.19	0.13	0.11	0.12	0.12	0.16	0.12	0.15
Timeouts	0	0	1	0	0	0	0	0	0	0
Wins	3	0	0	0	5	2	1	0	2	0

A first observation about these results is that the default settings consistently perform better than those learned by Piclasp, regardless of which heuristic is being used and regardless of whether parallel processing is used. This may be due to the fact that Piclasp does not support optimisation problems, or to the fact that we supplied it with relatively small training instances (because it needs to run a large number of experiments). For these reasons, Piclasp's training data may not have been representative enough for the actual problem.

Second, on a single processor, our domain-specific heuristic is almost always (in 10 of the 12 benchmarks) better than the default heuristic. In competitive multi-processor mode, the default heuristic has a slight edge (7 of the 12 benchmarks), whereas in split mode, the domain dependent heuristic has the edge (also 7 out of 12). When the default heuristic is used, competitive mode beats split mode (8 out of 12), whereas split mode always beats competitive mode for

the domain-dependent heuristic. Regardless of the heuristic, using all 8 processors instead of just a single processor leads to better results *if* the appropriate multi-processor strategy is chosen (e.g., dom-1 beats dom-comp, but loses to dom-split in 9 benchmarks).

We conclude from these results that the best variants are dom-split and def-comp. In the rest of our experiments, we use these two variants, together with dom-1, the best single-processor variant. We now compare these three variants with results from [16]. This article reports on three different approaches: a Constraint Programming (CP) approach, an Integer Programming (IP) approach (both using ILOG OPL Studio 3.7 as a solver), and the special-purpose search algorithm GBNS developed by its authors. The results reported for these three approaches are taken from [16] and were performed on an Intel(R) Xeon(TM) processor which ran at the same clock speeds of 3.2 GHz as the Intel(R) Core(TM) processor we used to test our own approach.

Table 2. Comparison for $n = 14$ between our methods (timeout: 10 min) and those of [16] (timeout: 3 h).

Benchmark	def-comp	IP [16]	dom-split	GBNS [16]	CP [16]	dom-1
14-5-3	0.14	0.13	0.13	**0.08**	0.18	0.15
14-6-3	**0.10**	0.15	0.11	0.11	0.16	0.11
14-7-3	0.07	0.15	**0.07**	0.12	0.09	0.07
14A-5-3	0.17	0.12	0.16	**0.10**	0.20	0.16
14A-6-3	0.10	0.13	0.12	0.12	0.20	**0.10**
14A-7-3	**0.06**	0.17	0.09	0.11	0.09	0.09
14B-5-3	0.14	0.12	0.13	**0.08**	0.19	0.15
14B-6-3	0.12	0.16	**0.11**	0.17	0.20	0.12
14B-7-3	**0.07**	0.21	0.09	0.09	0.10	0.09
14C-5-3	0.16	**0.12**	0.15	0.14	0.20	0.17
14C-6-3	0.15	0.10	**0.10**	0.14	0.24	0.13
14C-7-3	**0.11**	0.16	0.11	0.22	0.19	0.12
Avg	0.12	0.14	0.11	0.12	0.17	0.12
Wins	4	1	3	3	0	1

Table 2 shows results for $n = 14$. Our own approaches were benchmarked with a timeout of *10 min*, while the results reported for the approaches from [16] use a timeout of *3 h*. Even taking into account the fact that our multiprocessor variants use 8 processors instead of one, this still puts our approach at a disadvantage. Nevertheless, as can be seen in Table 2, it performs quite well. All of our variants beat Trick et al.'s CP solution on all benchmarks. Against Trick et al.'s IP solution, dom-split performs best, winning 7/12 benchmarks and tying 2; our single-processor variant and def-comp also win 7 but lose the others. Against

Table 3. Comparison for $n = 16$ between our methods and those of [16] (both timeouts: 3 h).

Benchmark	def-comp	CP [16]	GBNS [16]	dom-split	IP [16]	dom-1
16-7-2	0.27	0.34	**0.11**	0.26	0.14	0.26
16-7-3	0.18	0.32	0.17	**0.14**	NoSol	0.16
16-8-2	0.16	0.24	0.11	0.18	**0.10**	0.18
16-8-4	**UNSAT**	NoSol	NoSol	NoSol	NoSol	**UNSAT**
16A-7-2	0.25	0.36	**0.09**	0.22	0.17	0.23
16A-7-3	0.17	0.18	0.17	**0.15**	NoSol	0.15
16A-8-2	0.18	0.29	**0.10**	0.16	0.11	0.19
16A-8-4	**UNSAT**	NoSol	NoSol	NoSol	NoSol	**UNSAT**
16B-7-2	0.26	0.36	**0.11**	0.24	0.14	0.25
16B-7-3	0.18	0.25	NoSol	**0.18**	NoSol	0.18
16B-8-2	0.17	0.22	**0.11**	0.14	0.12	0.17
16B-8-4	**UNSAT**	NoSol	NoSol	NoSol	NoSol	NoSol
16C-7-2	0.21	0.23	**0.09**	0.21	0.18	0.21
16C-7-3	0.14	0.20	NoSol	**0.13**	0.18	0.14
16C-8-2	**0.11**	0.21	0.12	0.12	0.15	0.12
16C-8-4	**UNSAT**	NoSol	NoSol	NoSol	NoSol	**UNSAT**
Avg	0.19	0.27	0.12	0.18	0.14	0.19
Timeouts	0	4	6	4	7	1
Wins	5	0	6	4	1	3

their special-purpose GBNS algorithm, our single-processor variant wins 6 and ties 2 benchmarks, dom-split wins 5 and tries 3, while def-comp wins 7.

To judge also the performance for somewhat bigger instances, Table 3 shows experimental results for $n = 16$. Here, our own approaches were benchmarked with the same timeout of 3 h as in [16]. An entry of `UNSAT` in this table means that the system was able to report that no feasible solution exists; an entry of `NoSol` means that the system was unable to decide whether a feasible solution exists within the time limit. Also for $n = 16$, the CP solution does not win a single benchmark against any of our approaches. Against IP, our def-comp approach wins 9 out of 16 benchmarks, while dom-split wins only 5 and loses 7. Against GBNS, def-comp wins half of the benchmarks against GBNS and loses the other half. Our dom-split approach does worse, winning only 5 benchmarks against GBNS and losing 7.

In conclusion, our approach clearly outperform the CP approach and does slightly better than IP. It is also on par with the GBNS special-purpose algorithm. However, the same cannot be said for more recent special-purpose algorithms. The currently state-of-the-art the approach of [14] is able to find the optimal solution for each of the $n = 14$ benchmarks and most of the $n = 16$

ones. The best results reported in Tables 2 and 3 above are typically around 10% worse than those of [14].

7 Conclusions

A declarative approach allows computational problems to be solved without requiring the development of special-purpose algorithms. An advantage of such an approach is that the problem specification can easily be changed, by adding or replacing certain constraints. Moreover, developments in solver technology immediately improve the performance of declarative solutions for numerous different problems. A disadvantage is that the computational performance of declarative solvers often lags behind that of special-purpose algorithms, since it may take some time to lift improvements that were made to specific algorithm for one specific problem to the level of a generic reasoning tool. However, features such as domain-specific heuristics allow domain knowledge to be used to improve the performance of the solver.

In this paper, we examined a declarative solution for the Travelling Umpire Problem, which is a challenging optimisation problem that has recently received a lot of attention. As we have shown, it can be formulated in an elegant and modular way in the declarative ASP paradigm. Because the TUP as considered in the literature is only an abstraction of a real-life problem in Major League Baseball, the flexibility to easily add or change constraints is a useful feature for this application.

Using the state-of-the-art ASP solver Clasp, we found that the performance of our approach improves on that of previous declarative solutions and is on par with the first special-purpose algorithm published for this problem. However, it cannot match the performance of current special-purpose algorithms. This suggests that these algorithms use more advanced techniques which have not yet found their way into general-purpose ASP solvers. We believe the TUP might therefore be an interesting benchmark to guide future developments in ASP solver technology.

References

1. Alviano, M., Calimeri, F., Charwat, G., Dao-Tran, M., Dodaro, C., Ianni, G., Krennwallner, T., Kronegger, M., Oetsch, J., Pfandler, A., Pührer, J., Redl, C., Ricca, F., Schneider, P., Schwengerer, M., Spendier, L.K., Wallner, J.P., Xiao, G.: The fourth answer set programming competition: preliminary report. In: Cabalar, P., Son, T.C. (eds.) LPNMR 2013. LNCS, vol. 8148, pp. 42–53. Springer, Heidelberg (2013). doi:10.1007/978-3-642-40564-8_5
2. Calimeri, F., Gebser, M., Maratea, M., Ricca, F.: Design and results of the fifth answer set programming competition. Artif. Intell. **231**, 151–181 (2016)
3. Calimeri, F., et al.: The third answer set programming competition: preliminary report of the system competition track. In: Delgrande, J.P., Faber, W. (eds.) LPNMR 2011. LNCS, vol. 6645, pp. 388–403. Springer, Heidelberg (2011). doi:10.1007/978-3-642-20895-9_46

4. de Oliveira, L., de Souza, C., Yunes, T.: Improved bounds for the traveling umpire problem: a stronger formulation and a relax-and-fix heuristic. EJOR **2**, 592–600 (2014)
5. de Oliveira, L., de Souza, C., Yunes, T.: On the complexity of the traveling umpire problem. Theor. Comput. Sci. **562**, 101–111 (2015)
6. Denecker, M., Vennekens, J., Bond, S., Gebser, M., Truszczyński, M.: The second answer set programming competition. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 637–654. Springer, Heidelberg (2009). doi:10.1007/978-3-642-04238-6_75
7. Gebser, M., Harrison, A., Kaminski, R., Lifschitz, V., Schaub, T.: Abstract gringo. In: TPLP (2015)
8. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: the potsdam answer set solving collection. AI Communications **24**(2), 107–124 (2011)
9. Baral, C., Brewka, G., Schlipf, J. (eds.): LPNMR 2007. LNCS (LNAI), vol. 4483. Springer, Heidelberg (2007)
10. Gebser, M., Maratea, M., Ricca, F.: The design of the sixth answer set programming competition. In: Calimeri, F., Ianni, G., Truszczynski, M. (eds.) LPNMR 2015. LNCS, vol. 9345, pp. 531–544. Springer, Cham (2015). doi:10.1007/978-3-319-23264-5_44
11. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R., Bowen, K. (eds.) ICLP, pp. 1070–1080. MIT Press, Cambridge (1988)
12. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Coello, C.A.C. (ed.) LION 2011. LNCS, vol. 6683, pp. 507–523. Springer, Heidelberg (2011). doi:10.1007/978-3-642-25566-3_40
13. Toffolo, T., Van Malderen, S., Wauters, T.V., Berghe, G.: Branch-and-Price and Improved Bounds to the Traveling Umpire Problem. PATAT, York (2014)
14. Toffolo, T., Wauters, T., Van Malderen, S., Vanden Berghe, G.: Branch-and-bound with decomposition-based lower bounds for the traveling umpire problem. EJOR **3**, 932–943 (2015)
15. Trick, M.A., Yildiz, H.: Bender‘s cuts guided large neighborhood search for the traveling umpire problem. In: Van Hentenryck, P., Wolsey, L. (eds.) CPAIOR 2007. LNCS, vol. 4510, pp. 332–345. Springer, Heidelberg (2007). doi:10.1007/978-3-540-72397-4_24
16. Trick, M., Yildiz, H.: Benders' cuts guided large neighborhood search for the traveling umpire problem. Naval Res. Logistics (NRL) **8**, 771–781 (2011)
17. Trick, M., Yildiz, H.: Locally optimized crossover for the traveling umpire problem. EJOR **2**, 286–292 (2012)
18. Trick, M., Yildiz, H., Yunes, T.: Scheduling major league baseball umpires and the traveling umpire problem. Interfaces **3**, 232–244 (2012)
19. Wauters, T.: http://benchmark.gent.cs.kuleuven.be/tup/ Accessed 27 Oct 2016
20. Wauters, T., Van Malderen, S., Vanden Berghe, G.: Decomposition and local search based methods for the traveling umpire problem. EJOR **3**, 886–898 (2014)
21. Xue, L., Luo, Z., Lim, A.: Two exact algorithms for the traveling umpire problem. EJOR **3**, 932–943 (2015)