Online appendix for the paper

Combining Answer Set Programming and Domain Heuristics for Solving Hard Industrial Problems (Application Paper)

published in Theory and Practice of Logic Programming

Carmine Dodaro[1], Philip Gasteiger[2], Nicola Leone[1],

Benjamin Musitsch[2], Francesco Ricca[1], Kostyantyn Shchekotykhin[2]

[1]*Department of Mathematics and Computer Science, University of Calabria, Italy*
(*e-mail:* `{lastname}@mat.unical.it`)
[2]*Alpen-Adria-Universität Klagenfurt, Austria* (*e-mail:* `{firstname.lastname}@gmail.com`)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Appendix A Heuristics Development Example

In order to exemplify the usage of the infrastructure we report here the solution of the well-known Pigeonhole problem, whose ASP encoding is reported in Figure A 1. Albeit for this toy problem the solution is trivial, modern ASP solvers fail to recognize efficiently when an instance admits no solutions. It is easy to see that when the number of pigeons exceeds the number of holes, no solution can be found. Otherwise, a solution can be easily obtained by associating the *i-th* pigeon with the *i-th* hole.

A heuristic strategy based on this observation can be implemented using PYTHON as reported in Figure A 2. First, we initialize the global data structures `var`, `H`, and `P` for storing the association of atom names to a numeric identifier created by GRINGO, the set of holes, and the set of pigeons, respectively. The method `addedVarName` is called whenever a new variable `v` named `name` is added inside WASP. Here, we store the association between the variable identifier `v` and its name `name` and vice-versa. Moreover, we check whether the variable represents a pigeon or a hole by checking the name. If this is the case the ASP constant representing the pigeon or hole is added `P` or `H`, respectively.

After the parsing of the input program, the method `onFinishedParsing` is invoked. This method is allowed to return a list of variable that must be *frozen*, i.e. variables that must not be removed during the simplification step. In our example, all variables are frozen.

Later on, WASP searches for an answer set. During the computation, the method `choiceVars` is invoked whenever a choice is needed (ONCHOICEREQUIRED( )). This method may return:

- A literal representing the next choice (command #CHOOSE(ℓ)).
- A list of literals representing the next choices (command #CHOOSE(ℓ) repeated for all literals in the list).
- Special values representing other commands. In particular, `[4, 0]` is used to stop the computation returning INCONSISTENT (command #ADDCONSTRAINT($\leftarrow \sim\bot$)).

In our example, the method `choiceVars` first checks whether the holes are sufficient to host all

pigeons. If this is not the case it returns `[4,0]`. Otherwise, it returns a list of choices where the atoms `inHole(i,i)` ($i \in [1,\ldots,|P|]$) will be set to true.

Finally, the method `onChoiceContradictory` (event ONINCOCHOICE(ℓ)) is invoked whenever a previous choice lead to an inconsistency. In our case, this method performs no operation since none of the choices can be contradictory.

Appendix B Additional Plots

In this section, we present additional cactus plots comparing all off-the-shelf ASP solvers considered in our experiments with our best heuristic variant for each problem. In particular, Figures B 1 and B 2 report on the performance of the solvers on PUP instances employing encoding ENC1 and ENC2, respectively. Concerning ENC1, we observe that CLASP with 10 threads and CLASPFOLIO obtained similar performance solving 25 instances. All other systems solve 22 instances. Similar considerations hold also for ENC2, where the best versions are CLASP with 10 threads and CLASP with portfolio also solve 25 instances each. Looking at the VBS lines, we observe that all solvers behave similarly solving basically the same set of instances. However, in case of ENC1, VBS was able to solve one instance more than using ENC2 (`2-doublev-120.dl`). This is due to optimizations, like symmetry breaking rules, towards the grid and triple-like instances included in ENC2. On one hand, VBS with ENC2 required far less time to find solutions for grids and triples. On the other hand, due to these optimizations the performance on instances of the double and double-variant types was slightly lower than using ENC1. As a result, CLASP portfolio was able to solve one instance more, thus improving the overall performance of VBS. More detailed results can be found on the companion website `http://yarrick13.github.io/hwasp/`.

Concerning CCP, the performance of the solvers is reported in Figure B 3. Here, all considered solvers solve at most 5 instances.

As a general comment we observe that multi-threaded versions exploiting 10 times more hardware resources and several heuristics are slightly better than single-threaded alternatives. Nonetheless, our heuristic variants are faster than all other alternatives.

```
# Input:
# * a set of pigeons P={1,...,n}, defined by means of the predicate pigeon
# * a set of holes   H={1,...,m}, defined by means of the predicate hole
pigeon(1) ←          hole(1) ←
   ⋮                    ⋮
pigeon(n) ←          hole(m) ←

# Guess an assignment
inHole(p,h) ← not outHole(p,h) ∀p ∈ P, ∀h ∈ H
outHole(p,h) ← not inHole(p,h) ∀p ∈ P, ∀h ∈ H

# A hole contains at most one pigeon
← inHole(p_i,h), inHole(p_j,h) ∀p_i, p_j ∈ P | i ≠ j, ∀h ∈ H

# A pigeon is assigned to at most one hole
← inHole(p,h_i), inHole(p,h_j) ∀h_i, h_j ∈ H | i ≠ j, ∀p ∈ P

# A pigeon must be in some hole
inSomeHole(p) ← inHole(p,h) ∀p ∈ P, ∀h ∈ H
← not inSomeHole(p) ∀p ∈ P
```

Fig. A 1. ASP encoding of Pigeonhole.

```python
# global data structures
var = {1: 'false', 'false': 1}
P = [] # list of all pigeon-constants
H = [] # list of all hole-constants

def addedVarName(v, name):
    #invoked when WASP parses the atom table of the gringo numeric format
    global var, H, P
    var.update({v: name, name: v})
    if name.startswith("pigeon"):
        P.append(name[7:-1])
    if name.startswith("hole"):
        H.append(name[5:-1])

def onFinishedParsing():
    # disable simplifications of variables
    return [v for v in var.keys() if isinstance(v, int)]

def choiceVars(): #event onChoiceRequired, invoked when a choice is needed
    global var, H, P
    if len(P) > len(H):
        return [4, 0] # force incoherence
    # assign pigeon i to hole i
    return [var["inHole(%s,%s)" % (i, i)] for i in range(1, len(P))]

def onChoiceContradictory(choice): #event onIncoChoice
    pass # no choice can be contradictory
```

Fig. A 2. Pigeonhole heuristic in PYTHON.

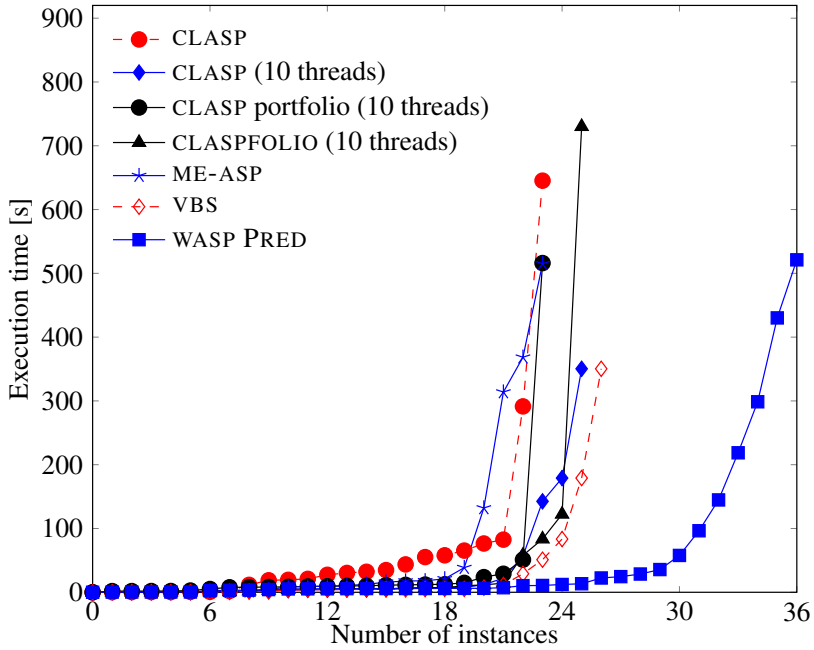

Fig. B 1. Comparison of all solvers on PUP instances (ENC1)

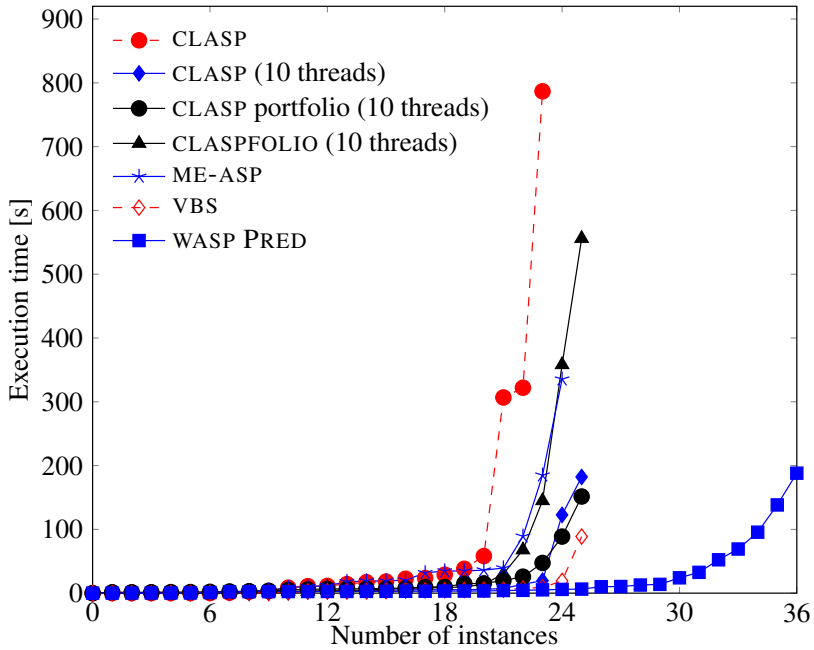

Fig. B 2. Comparison of all solvers on PUP instances (ENC2)

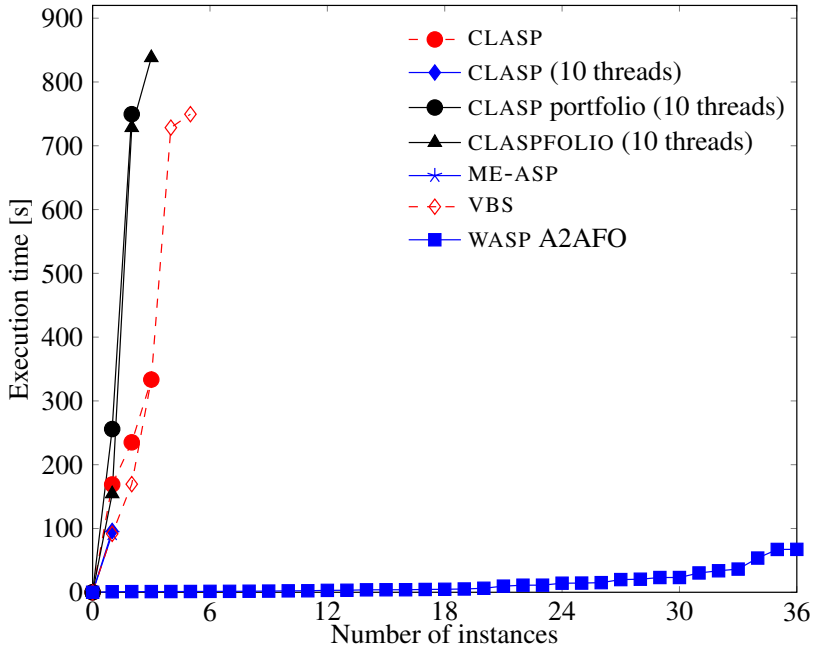

Fig. B 3. Comparison of all solvers on CCP instances