Practical and Methodological Aspects of the Use of Cutting-Edge ASP Tools

Marcello Balduccini[1] and Yulia Lierler[2]

[1] Eastman Kodak Company
marcello.balduccini@gmail.com
[2] University of Kentucky
yuliya@cs.utexas.edu

Abstract In the development of practical applications of answer set programming (ASP), encodings that use well-established solvers such as CLASP and DLV are sometimes affected by scalability issues. In those situations, one can resort to more sophisticated ASP tools exploiting, for instance, incremental and constraint ASP. However, today there is no specific methodology for the selection or use of such tools. In this paper we describe how we used such cutting-edge ASP tools on challenging problems from the *Third Answer Set Programming Competition*. We view this paper as a first step in the development of a general methodology for the use of advanced ASP tools.

Keywords: answer set programming, solvers, constraint ASP, incremental ASP.

1 Introduction

The *Third Answer Set Programming Competition – 2011* [1] (ASPCOMP) included a *Model and Solve* track. Within this track the teams were free to choose a specific declarative solver and modeling technique for each problem. Answer set programming (ASP) solvers were the primary focus. Nowadays, there are a number of well-established ASP solvers such as CLASP [6], DLV [8], and cutting-edge solvers based on constraint and incremental ASP (resp., CASP, IASP), such as EZCSP [3] and ICLINGO [5]. Well-established solvers are robust and their use relies on a well-understood programming methodology. On the other hand, in some circumstances the encodings for these systems have scalability issues. The extensions of ASP implemented by the solvers for CASP and IASP aim at overcoming some of these issues. However, today there is no specific methodology for the formalization of knowledge with such new tools, or even for the selection of a suitable tool given the features of a domain.

In this paper we describe how we used CASP and IASP tools to tackle four challenging ASPCOMP benchmarks (*Weight-Assignment, Reverse-Folding, Hydraulic-System-Planning, and Airport-Pickup*). Throughout our description, we provide methodological considerations, both from the perspective of tool selection and of knowledge representation. Although the discussion in this paper is still oriented towards the specific problems we solved, we view this effort as a first, necessary step towards a general methodology for the use of advanced ASP tools.

Our decisions with respect to tool selection and modeling techniques were based on problem statement analysis and performance assessments that we conducted on the

training instances available before the competition. Because the number of training instances was rather small – ranging from 5 to 7 – we could not carry out a thorough pre-competition performance assessment. Nonetheless, the evaluation gave us some evidence [4] of the performance yielded by our encodings. After the competition we conducted a post-competition performance assessment described in Section 7.

The structure of this paper is as follows. We begin with a short introduction on ASP, CASP and IASP. Sections 3, 4, 5, and 6 provide the problem statements and the specifications of the encodings for the Weight-Assignment, Reverse-Folding, Hydraulic-System-Planning, and Airport-Pickup benchmarks, respectively. In Section 7 we discuss performance. In the final section we draw conclusions.

2 Background

Because of space considerations, in this section we only provide a short introduction on ASP, CASP and IASP, and refer the reader to [7], [3] and [5], respectively for the syntax and semantics of the corresponding languages.

ASP is a declarative programming paradigm based on the answer set semantics of logic programs [7]. The idea of ASP is to represent a given problem by a program whose answer sets correspond to solutions. A common programming methodology is to design two main parts of a program: *generate* and *test*. The former defines a collection of answer sets, seen as potential solutions. The latter consists of the rules that remove the non-solutions. To distinguish the language in [7] from its extensions, we talk about *pure* ASP, programs, and rules.

CASP extends the syntax and semantics of ASP with constraint processing elements. It allows for new modeling features and novel computational methods that combine traditional ASP procedures with constraint satisfaction (CSP) and constraint logic programming (CLP) algorithms. CASP is especially useful in domains that pose constraints over large numerical values. In such cases, grounding often becomes a bottleneck in the pure ASP approach. EZCSP is an inference engine for CASP that allows a lightweight integration of ASP and constraint programming. In the EZCSP terminology, an *extended answer set* of a CASP program Π, is a pair consisting of an answer set of Π where some of the atoms encode CSP constraints, and of a solution to these CSP constraints. Given a program Π, the EZCSP solver computes one or more of Π's extended answer sets. The solver combines off-the-shelf ASP (e.g., CLASP) and CLP solvers (e.g., BPROLOG, `http://www.probp.com/`). The architecture is such that first the ASP solver is used to find an answer set A of a given CASP program Π. Then the CSP constraints encoded by A are evaluated by the constraint solver. If the solver determines that the constraints from A are not satisfiable, another answer set is computed and the process repeats. Otherwise, A and a solution found by the constraint solver form an extended answer set. If Π is a pure ASP program then EZCSP behaves as its underlying ASP solver. Conversely, Π may also be a direct encoding of a CSP theory, and in this case EZCSP behaves as its underlying constraint solver.

In certain domains, a numerical parameter can be identified that reflects the size or complexity of a candidate solution. IASP extends pure ASP by allowing one to take advantage of such a parameter (the *growth parameter*). The programmer is given means

to denote rules that are independent of the growth parameter (the *base*), rules whose grounding should be computed incrementally in dependence of the value of the parameter (the *cumulative part*), and rules that should be grounded anew for each different value of the parameter considered (the *volatile part*). An incremental answer set solver such as ICLINGO first attempts to find a solution for a minimum value of the growth parameter. If unsuccessful, it iteratively (1) increments this value, incrementally grows the grounding of the cumulative part of the program, (2) re-grounds the volatile part, and (3) checks again for a solution.

3 Weight-Assignment Benchmark

In the Weight-Assignment Benchmark, a binary tree with n leaves is considered, such that (1) the leaves are pairs of integers $\langle weight, cardinality\rangle$; (2) the right child of an inner node is a leaf; (3) each inner node is a pair $\langle color, weight\rangle$, where $color$ is green, red, or blue; (4) the inner nodes are numbered from 1 to $n-1$; node $n-1$ is the root node; the left child of each inner node i is inner node $i-1$. The weight of an inner node k is computed as follows: (1) if the color of k is green, then $weight(k)$ is the sum of the weight and cardinality of k's right child; (2) if the color of k is red, then its weight is the sum of the weight of its right and left children; (3) if the color of k is blue, then its weight is the sum of the cardinality of its right child and of the weight of its left child. The task is to verify that there is a tree formed by the given leaves in accordance with (1-4) so that the total weight of this tree – the sum of the weights of its inner nodes – is less than or equal to a given integer *maximum weight*. Problem instances are specified by the relations of the form $leafWeightCard(l, w, c)$, $num(n)$, and $maxWeight(mv)$, where l is a name of the leaf $\langle w, c\rangle$, n is a number of leaves, and mv is a maximum weight. More detailed descriptions of the Weight-Assignment benchmark and also of other benchmarks discussed in this paper are given on the ASPCOMP website [1].

Because of the abundance of constraints over numerical values, i.e., weights and cardinalities of the leaves and inner nodes, this benchmarks lends itself to being solved using EZCSP.

Hybrid ASP-CSP Encoding: let n and mv denote a number of leaves and a maximum weight in a given weight-assignment problem instance, respectively. We say that a leaf l occurs at position $1 \leq p < n$ in the tree if it is the right child of an inner node p. Furthermore, a position of an inner node is identified with a number associated with it, i.e., $1 \ldots n-1$. A leaf occurs at position 0 if it is the left child of an inner node 1. We model the assignment of a leaf to position p by the relation $assignedLeafPos(l, p)$ that denotes that a leaf with the name l is assigned a position p. Set of rules (1) below states that each leaf is assigned a unique position. Relation $innerNodeColor(k, c)$ denotes the fact that inner node k is assigned a color c. Rule (2) states that each inner node (identified by its position) is assigned a single color.

$$1\{assignedLeafPos(L, P) : position(P)\}1 \leftarrow leafWeightCard(L, W, C). \quad (1)$$
$$\leftarrow assignedLeafPos(L, P),\ assignedLeafPos(L', P),\ L \neq L'.$$
$$1\{innerNodeColor(P, C) : color(C)\}1 \leftarrow position(P), P \neq 0. \quad (2)$$

The weight of an inner node k is modeled by a CSP variable $weight(k)$, whose value ranges from 0 to mv. In order to simplify the encoding of the constraints, we use CSP

variable $weight(0)$ to denote the weight of the leaf at position 0. The corresponding rules are:

$cspvar(weight(K), 0, MV) \leftarrow num(N), K = 0..N-1,\ maxWeight(MV).$

$required(weight(0) = W) \leftarrow assignedLeafPos(L, 0),\ leafWeightCard(L, W, C).$

The first rule declares the CSP variables of the form $weight(k)$. The other rule encodes a CSP constraint that determines the value of variable $weight(0)$ to be the weight of the leaf assigned position 0. The constraints on the weights of the inner nodes are encoded by statements such as:

$$\begin{aligned} required(weight(P) = W + weight(P')) \leftarrow \\ position(P),\ P \neq 0,\ P' = P-1,\ innerNodeColor(P, red), \\ assignedLeafPos(L, P),\ leafWeightCard(L, W, C). \end{aligned} \tag{3}$$

To compute the total weight of a tree, we introduce a set of auxiliary CSP variables of the form $innerWeight(k)$, where k ranges from 1 to $n-1$. For every k in that range, variable $innerWeight(k)$ equals $weight(k)$:

$cspvar(innerWeight(K), 0, MV) \leftarrow num(N),\ K = 1..N-1,\ maxWeight(MV).$

$required(innerWeight(K) = weight(K)) \leftarrow num(N),\ K = 1..N-1.$

$required(sum([innerWeight/1], \leq, MV)) \leftarrow maxWeight(MV).$

The last rule encodes a CSP constraint stating that the sum of the weights of the inner nodes of the tree must be less than or equal to mv. We denote the program consisting of the rules discussed so far by $\Pi_1(WA)$.

Encoding Analysis: in program $\Pi_1(WA)$, the generate part consists of the rules in (1) and (2). The rest of the rules form the test part. Note that generation is formed by pure ASP rules whereas testing is formulated using rules that contain CSP variables. Recall the general architecture of the EZCSP system discussed in Section 2. It is not difficult to see that in the worst-case scenario (for instance, when the problem is unsatisfiable) EZCSP will generate and evaluate every possible combination of leaf-position/inner node-color assignments during its search process. To avoid such behavior we restate the generate part of the program so that the CSP solver of the EZCSP system is responsible for both generate and test. Thus the encoding we discuss next can be viewed as a CSP formalization of the weight-assignment problem by means of a CASP language. We denote this encoding by $\Pi_2(WA)$.

CSP Formalization by Means of CASP: we begin by modeling the assignment of a leaf to a position p by the CSP variable $assignedLeaf(p)$. Since CSP variables have numerical values, we map the name l of a leaf $\langle w, c\rangle$ (given by $leafWeightCard(l, w, c)$) to an integer id ranging from 1 to n and add an auxiliary fact $leafId(l, id)$ to a program. The EZCSP declaration of $assignedLeaf(p)$ is:

$$cspvar(assignedLeaf(P), 1, N) \leftarrow position(P),\ num(N). \tag{4}$$

The fact that a leaf can only be assigned one position is compactly enforced by means of a global constraint $all\_different$, encoded by

$$required(all\_different([assignedLeaf/1])). \tag{5}$$

where the expression $[assignedLeaf/1]$ denotes the list of the CSP variables formed by function symbol $assignedLeaf$ with arity 1. Rules (4) and (5) are the counterparts of (1) in $\Pi_1(WA)$. The statement

$$cspvar(innerNodeColor(P), 0, 2) \leftarrow position(P). \tag{6}$$

declares a CSP variable $innerNodeColor(k)$ for each inner node k; the value of the variable denotes the color assigned to k that ranges between 0 (representing color red) and 2 (representing blue). The association between a color and its identifier is encoded by a set of facts of the form $colorId(c, id)$, where c is red, blue or green, and id is its identifier. The declaration of $innerNodeColor(k)$ is the counterpart of rule (2). As in Π_1(*WA*), the weight of an inner node k is modeled by a CSP variable $weight(k)$. The variable declaration remains the same, but the encoding of the requirements on $weight(k)$ is different. For instance, rule (3) becomes:

$$\begin{aligned} required((innerNodeColor(P) = REDID \ \wedge \ assignedLeaf(P) = ID) \rightarrow \\ weight(P) = W + weight(P')) \leftarrow \\ position(P),\ P \neq 0,\ P' = P - 1, \\ colorId(red, REDID),\ leafId(L, ID),\ leafWeightCard(L, W, C). \end{aligned}$$

The rules for $innerWeight$ are reformulated similarly. In order to improve performance, the encoding also contains constraints that provide bounds for the value of the weight of an inner node *independently of the color of the node*, such as:

$$\begin{aligned} required(assignedLeaf(P) = ID \rightarrow weight(P) \geq \\ min(W + C, min(W + weight(P'), C + weight(P')))) \leftarrow \\ position(P),\ P \neq 0,\ P' = P - 1,\ leafId(L, ID),\ leafWeightCard(L, W, C). \end{aligned}$$

4 Reverse-Folding Benchmark

In the Reverse-Folding benchmark, one manipulates a sequence of n pairwise connected segments located on a 2D plane in order to take the sequence from the initial configuration to the goal configuration specified. The ordering of the sequence and the fact that the segments are connected to each other allows one to label each end point of a segment either as a starting point or as an ending point. All segments have unary length, and are parallel either to the x-axis or to the y-axis. In the initial configuration, the segments are parallel to the y-axis and oriented so that the sequence extends in the direction of the positive y-axis. The sequence is manipulated by rotating a segment around its starting point by 90 degree (in either direction). This action is called pivot move. A pivot move on a segment causes the segments that follow it to rotate around the same center of rotation. Concurrent pivot moves are prohibited. At the end of a pivot move, the segments in the sequence must not intersect. In the Reverse-Folding problem, one is given the number n of segments (relation $length$), the goal configuration (relation $fold(i, x, y)$, where $1 \leq i \leq n$ and x, y are the coordinates of the i^{th} starting point, or of the ending point of the last segment for $i = n$), and an integer t (relation $time$). The task is to find a sequence of exactly t pivot moves, which produces the goal configuration from the initial configuration, satisfying the constraints cited above. A solution is encoded as a set of atoms of the form $pivot(t, i, r)$, saying that the t^{th} pivot move rotates the i^{th} segment either clockwise ($r = clock$) or counterclockwise ($r = anticlock$).

Simple Encoding: in writing an encoding that solves this benchmark, the first thing that became apparent is that a minimum number of necessary pivot moves can be inferred directly by observing the structure of the goal configuration. If two segments are at an angle in the goal configuration, it is not difficult to prove that every solution to the problem instance must contain a pivot move that rotates the second segment of

the pair. In order to infer such moves, we first define a relation $segDirection(i, d, o)$, which intuitively states that the i^{th} segment in the goal sequence has direction d and orientation o. For example, the rules for the segments parallel to the x-axis are:

$$segDirection(I, horiz, plus) \leftarrow X2 > X1,\ fold(I, X1, Y),\ fold(I+1, X2, Y).$$
$$segDirection(I, horiz, minus) \leftarrow X2 < X1,\ fold(I, X1, Y),\ fold(I+1, X2, Y).$$

Next, we define relation $foldDirection(i, d)$, intuitively saying that in the goal configuration the i^{th} segment is aligned with its predecessor ($r = none$), or rotated clockwise or counterclockwise with respect to it ($r = clock$ and $r = anticlock$, respectively). The rules for $r \in \{none, clock\}$ are:

$$foldDirection(I, none) \leftarrow segDirection(I-1, D, O),\ segDirection(I, D, O).$$
$$foldDirection(I, clock) \leftarrow clockFold(D1, O1, D2, O2),\ segDirection(I-1, D1, O1),\ segDirection(I, D2, O2).$$
$$clockFold(vert, plus, horiz, plus).\ clockFold(horiz, plus, vert, minus).\ \ldots$$

Finally, relation $requiredFold(i, r)$ says that the i^{th} segment must be rotated clockwise or counterclockwise:

$$requiredFold(I, R) \leftarrow R \neq none,\ foldDirection(I, R).$$

In most cases, performing the pivot moves beginning from the end of the sequence produces a solution. In this case, the pivot moves can be determined by the rules:

$$pivot(1, I, R) \leftarrow first(I),\ requiredFold(I, R).$$
$$pivot(T1, I1, R1) \leftarrow pivot(T, I2, R2),\ T1 = T+1,\ next(I1, I2),\ requiredFold(I1, R1),\ requiredFold(I2, R2).$$

where $first(i)$ and $next(i_1, i_2)$ enumerate the segments that are to be rotated, beginning from the one closest to the end of the sequence. Because a solution to the Reverse-Folding problem is required to contain exactly the specified number of moves, it may happen that extra, irrelevant moves need to be generated. This can be achieved by alternating clockwise and counterclockwise rotations of segment 1:

$$pivot(T1, 1, clock) \leftarrow numRequiredFolds(R),\ time(T),\ T1 > R,\ T1 \leq T,\ (T1 - R) \bmod 2 = 1.$$

Relation $numRequiredFolds(r)$ says that r required folds were identified in the goal configuration. The rule for $anticlock$ is similar. Next, we ensure that there are no overlapping segments during the execution of the moves. To achieve this, we project the effects of each move on the segments and check for an overlap. To reduce the size of the grounding, we consider separately the effects of the rotations on the x and y coordinates of the end points of the segments. The information is encoded by $foldx(t, i, p)$ and $foldy(t, i, p)$, saying that the x (resp., y) coordinate of the i^{th} end point before move t is p. The effect of a move on the x coordinate of a segment is encoded by:

$$\begin{aligned}
&foldx(T1, I, Y - Y1 + X1) \leftarrow foldy(T, I, Y),\ pivot(T, I1, clock),\ I \geq I1,\ T1 = T+1,\ foldx(T, I1, X1),\ foldy(T, I1, Y1).\\
&foldx(T1, I, Y1 - Y + X1) \leftarrow foldy(T, I, Y),\ pivot(T, I1, anticlock),\ I \geq I1,\ T1 = T+1,\ foldx(T, I1, X1),\ foldy(T, I1, Y1).\\
&foldx(T1, I, X) \leftarrow foldx(T, I, X),\ pivot(T, I1, R),\ I < I1,\ T1 = T+1.
\end{aligned} \tag{7}$$

The first two rules state the effect of clockwise and counterclockwise rotations on the segments that follow the point where the rotation is applied. The last rule states that

the x coordinate of the other end points is unchanged. The definition of $foldy$ is similar. The following denial states that overlaps are not allowed to occur:

$$\leftarrow foldx(T, I1, X1),\ foldy(T, I1, Y1),\ foldx(T, I2, X1), foldy(T, I2, Y1),\\ I1 < I2,\ pivot(T-1, I3, R),\ I2 > I3.$$

The two inequalities in the denial are aimed at reducing the size of the grounding, the former by exploiting symmetry considerations, and the second by preventing the denial from considering segments that were not affected by the pivot move. Finally, relations $foldx$ and $foldy$ are used to ensure that the goal configuration is eventually reached:

$$\begin{array}{l}\leftarrow time(T),\ T1 = T+1,\ X1 \neq X2,\ foldx(T1, I, X1),\ fold(I, X2, Y2).\\ \leftarrow time(T),\ T1 = T+1,\ Y1 \neq Y2,\ foldy(T1, I, Y1),\ fold(I, X2, Y2).\end{array} \quad (8)$$

The program consisting of the rules discussed so far will be denoted by $\Pi_1(RF)$.

Encoding Analysis: Unfortunately, the presence of the pivot moves identified by $\Pi_1(RF)$ is a necessary, but not always sufficient, condition to find a solution. In some cases, executing the pivot moves beginning from the end of the sequence of segments causes some segments to overlap, but the moves can be re-ordered so that no overlap exists. In particular, it is often possible to find a solution by postponing one (suitable) pivot move to the end of the sequence of moves. We call this the *delayed-move* case. (To keep this presentation simple, other cases are not discussed.)

The delayed-move case can be handled by adding a choice rule for the selection of one delayed move and modifying the definition of relation $pivot$ so that the delayed move is executed at the end of the sequence of moves. One such choice rule is:

$$0\{\ delayed(I) : requiredFold(I, D)\ \}1.$$

Let $\Pi_2(RF)$ denote the modified program. The computation for $\Pi_2(RF)$ is substantially slower than the computation for $\Pi_1(RF)$, with the performance of the grounding process particularly affected. In $\Pi_2(RF)$ the grounder does not handle efficiently the rules involving $foldx$ and $foldy$, whose arguments have rather large numerical domains. Recall that the definitions of $foldx$ and $foldy$ rely on relation $pivot$, whose definition in $\Pi_2(RF)$ differs from the one in $\Pi_1(RF)$. Hence, we created a variant $\Pi_3(RF)$ of $\Pi_2(RF)$ that takes advantage of CASP capabilities of EZCSP by encoding constraints on $foldx$ and $foldy$ using CSP, such as:

$$required(foldx^{\gamma}(T1, I) = foldy^{\gamma}(T, I) - foldy^{\gamma}(T, I1) + foldx^{\gamma}(T, I1)) \leftarrow\\ pivot(T, I1, clock),\ T1 = T+1,\ I \geq I1.$$

5 Hydraulic-System-Planning Benchmark

In the Hydraulic-System-Planning benchmark, a hydraulic system is viewed as a directed graph G. The nodes of G represent tanks, jets, and junctions. Tanks are either empty or full. Each link between nodes is labeled by a valve. A valve can be opened (by action $switchon$). Valves that are *stuck* cannot be opened. A node of G is called *pressurized* in state S if it is a full tank or if there exists a path from some full tank to this node such that all valves on the edges of this path are open. Furthermore, no path connecting two tanks exists and every jet is connected to at least one tank. An input for this benchmark consists of a graph G, a specification of which tanks are full and which

valves are stuck (all valves are initially closed), and a set of *goal jets*. The goal is to find a shortest sequence of $switchon$ actions to pressurize the goal jets. In the sequence, no actions can be executed concurrently.

The challenge in this benchmark is that the length of the sequence of actions must be minimized. From a methodological standpoint, we approached the problem by first writing a pure ASP encoding, and then addressing its performance by transforming it into an ICLINGO program. For later reference, we label various sets of rules as we introduce them. We define an important notion of *viable path* as a path in G such that no valve along the path is stuck. Relation $viablePath(j, n)$ formalizes this notion recursively, restricting it to the goal jets for efficiency:

$$viablePath(J, J) \leftarrow goal(J).$$
$$viablePath(J, N') \leftarrow goal(J),\ viablePath(J, N),\ link(N', N, V),\ \text{not}\ stuck(V).$$

The following rules ensure that there is a viable path to a full tank for every goal jet:

$$canPressurize(J) \leftarrow goal(J),\ full(T),\ viablePath(J, T).$$
$$\leftarrow goal(J),\ \text{not}\ canPressurize(J).$$

Let $\Pi_1(HP)$ denote all of the rules above. Next, we address the planning task in two steps. In the first step we find the length of the shortest viable paths between each goal jet and a full tank, and in the second step we determine a sequence of actions that opens the paths of the given length. We begin by defining the notion of reachability in a given number of steps, which again we restrict to goal jets for performance:

$$\begin{aligned} &reachable(J, J, 0) \leftarrow goal(J).\\ &reachable(J, N', S) \leftarrow\\ &\qquad goal(J),\ reachable(J, N, S-1),\ link(N', N, V),\ \text{not}\ stuck(V). \end{aligned} \tag{9}$$

Using this relation, we can now define the notion of a *pressure path* of length k between goal jet j and full tank t, i.e. a viable path of length k between j and t:

$$pressurePath(J, T, S) \leftarrow goal(J),\ full(T),\ reachable(J, T, S). \tag{10}$$

We denote the set of rules (9) and (10) by $\Pi_2(HP)$. Next we describe the set of rules that form $\Pi_3(HP)$. The length of the shortest paths from goal jet j to any full tank is defined by:

$$\begin{aligned} &shortestPath(J, Len) \leftarrow\\ &\qquad goal(J),\ Len = \#\min[\, pressurePath(J, T, L) = L : full(T)\,]. \end{aligned}$$

Note that there may be multiple shortest paths for a goal jet. Therefore, we determine a single shortest path for each jet. We begin by defining the notion of valves that *can be possibly* used to open a shortest path for a given jet. We encode this notion recursively using relation $poss\_use\_valve(j, n, v, s)$, which states that at the end of the path from j to node n of length s, valve v can be possibly used:

$$\begin{aligned} poss\_use\_valve(J, N, V, S-1) \leftarrow\ & goal(J),\ shortestPath(J, S),\ full(T), link(T, N, V),\\ & reachable(J, T, S),\ reachable(J, N, S-1).\\ poss\_use\_valve(J, N2, V2, S-1) \leftarrow\ & goal(J),\ poss\_use\_valve(J, N1, V1, S),\\ & reachable(J, N2, S-1),\ link(N1, N2, V2). \end{aligned}$$

The recursion intuitively enumerates the valves moving from a tank towards a goal jet. The first rule encodes the base case and says that if the shortest paths for jet j have

length s and a full tank t is reachable from j in s steps, then for any node n connected to t and reachable from j in $s-1$ steps, the connecting valve v can be used at the end of the path from j to n. The second rule states that, if valve v_1 can be possibly used at the end of the path from j to n_1 of length s, then for any node n_2 reachable from j in $s-1$ steps and directly connected to n_1 by valve v_2, v_2 can be possibly used at the end of the path to n_2 of length $s-1$.

The selection of valves to be used is also performed recursively. We begin by considering, for each jet j, all paths of length 0. We select exactly one valve among the valves that can be possibly used at the end of each of those paths:

$$1\{\ use\_valve(J,N,V,0) : poss\_use\_valve(J,N,V,0)\ \}1 \leftarrow goal(J).$$

Next, given the decision to use valve v at the end of the path from j to n of length s, we identify the node, n', connected to n by v, and select exactly one valve among the ones that can be possibly used at the end of the path from j to n':

$$\begin{aligned} 1\{\ use\_valve(J,N2,V2,S+1) &: poss\_use\_valve(J,N2,V2,S+1) \\ &: link(N2,N1,V1)\ \ : \text{not } tank(N2)\ \}1 \leftarrow \\ goal(J),\ shortestPath(J,MS),\ use\_valve(J,N1,V1,S),\ S &< MS-1. \end{aligned}$$

Finally, we generate the corresponding $switchon$ actions. Because the actions cannot be executed concurrently, we produce a global ordering of the actions. This is achieved by, first, ordering the goal jets (in lexicographic order according to their name). Second, we schedule the execution of the actions for the first jet, followed by the actions for the second jet, and so on. We define relation $num\_prevActions(j,n)$, which states that n is the number of actions to be executed before the first action for goal jet j takes place:

$$\begin{aligned} num\_prevActions(J,NP) \leftarrow\ & goal(J), \\ & NP = \#sum[\ shortestPath(J1,N) = N : J1 < J\ ]. \end{aligned}$$

At this point, the $switchon$ actions for a jet j are scheduled to progressively open the path beginning from the tank that has been selected to feed j:

$$\begin{aligned} switchon(V,S-LS-1+NP) \leftarrow\ & goal(J),\ shortestPath(J,S), \\ & num\_prevActions(J,NP),\ use\_valve(J,N,V,LS). \end{aligned}$$

This concludes the description of $\Pi_3(HP)$.

Encoding Analysis: It is not difficult to see that the program $\Pi(HP)$ consisting of $\Pi_1(HP)$ – $\Pi_3(HP)$ may not scale well. As the size of the graph grows, the number of possible paths of arbitrary length may grow dramatically, leading to an explosion in the grounding. However, because the goal is to find a shortest path for each goal jet, the search performed by $\Pi(HP)$ could be intuitively done in an incremental fashion. Among the ASP tools available, ICLINGO[5] offers a simple way to deal with programs that involve an incremental search, and program $\Pi(HP)$ lends itself to being extended to exploit the features of ICLINGO.

IASP Encoding: First, we identify the set $\Pi'_b(HP)$ of rules that define the base of the program. $\Pi'_b(HP)$ consists of $\Pi_1(HP)$ together with the first rule in (9). The presence

of $\Pi_1(HP)$ is particularly important from the point of view of performance, because it allows to identify a problem instance that has no solution without performing any iteration of the search. Let s denote the growth parameter. The cumulative part, $\Pi'_c(HP)$, of the program includes a number of elements. First, $\Pi'_c(HP)$ includes a modification of the second rule in (9) and rule (10) where these two rules contain an additional condition $S = s$. This allows us to restrict the grounding of the rules to only the paths of the length considered by the current iteration of the search. *The semantics of the rules changes so that now they define, respectively, reachability in exactly s steps and the presence of a pressure path of length s. The overall meaning of the relations remains unchanged because the cumulative part of an* ICLINGO *program is implicitly quantified over all of the possible values of the growth parameter.*

Next, we add to $\Pi'_c(HP)$ rules aimed at detecting when the length of the shortest paths for all goal jets can be computed. This detection was not needed in the pure ASP program, but is used here to terminate the iterations of the search process:

$$\neg orphan(J, s) \leftarrow goal(J),\ S \leq s,\ pressurePath(J, T, S).$$
$$orphans(s) \leftarrow goal(J),\ \text{not}\ \neg orphan(J, s).$$
$$all\_jets\_fed(s) \leftarrow \text{not}\ orphans(s).$$

The key notion defined by the above rules is that of an *orphan* goal jet. A goal jet j is orphan of rank s if no pressure path of length s or less exists for j. The second rule determines if there are still orphans of rank s. The last rule states that $all\_jets\_fed(s)$ holds if no orphans of rank s exist.

Finally, $\Pi'_c(HP)$ includes $\Pi_3(HP)$ modified by adding to each rule the condition $all\_jets\_fed(s)$. This modification ensures that the rules are considered only if pressure paths of length s or less exist for every goal jet.

The volatile part $\Pi'_v(HP)$ of the program contains the denial $\leftarrow\ orphans(s)$, which states that it is impossible for the iterative search to terminate at step s if orphans of rank s exist. This constraint forces the iterative search to continue until pressure paths have been found for every goal jet. Once these have been found, the rules in $\Pi'_c(HP)$ select a shortest path for each goal jet and determine a suitable sequence of $switchon$ operation. By $\Pi'(HP)$ we denote the union of $\Pi'_b(HP)$, $\Pi'_c(HP)$, and $\Pi'_v(HP)$. Answer sets of $\Pi'(HP)$ encode solutions to the problem instances.

6 Airport-Pickup Benchmark

In the Airport-Pickup benchmark, one must solve resource-based planning problems that involve objects moving between locations. More precisely, a city is represented by a weighted undirected graph G. The nodes of G represent locations where exactly two of them are airports. Some locations may contain gas stations. The arcs of G represent direct connections between the locations and are labeled with an integer corresponding to the amount of gas required to travel between them. The problem also involves a set of vehicles and a set of passengers. A vehicle can initially be at any location, and can travel from its current location, l, to any location connected to l as long as it has enough gas. A problem instance specifies the amount of gas in each vehicle originally. Each passenger is initially located at an airport, and his goal is to reach the other airport.

Passengers can move between locations only by vehicle. Vehicles can pick up and drop off passengers, but only one passenger at a time can ride a vehicle. Finally, vehicles can fill their tanks at a gas station. The goal is to find a sequence of actions that takes each passenger to its goal destination.

This benchmark is interesting because the large size of the corresponding search space makes it difficult to solve it efficiently using a single call to a solver. In our initial evaluation we could not find any such "monolithic" encoding that would scale to the training instances provided for ASPCOMP. For this reason, we decided to adopt an approach in which the problem is divided into sub-problems, and multiple calls to solvers are used. It is important to stress that this approach, although not frequently discussed in the literature, can be extremely useful in practical applications of ASP.

Our solution of the Airport-Pickup benchmark is based on an architecture consisting of a main module, tackling the overall search problem, and of a number of auxiliary modules, to which the main module delegates the solution of various sub-problems. This allows us to limit the size of the grounding of the programs, and at the same time makes it possible to use the language/solver best suited for each module. The main module, $\Pi_1(AP)$, employs an extension of ASP developed for controlling the interactions among modules [2]. To keep the presentation simple we abstract from the technical details of the control structure, and describe $\Pi_1(AP)$ as a pure ASP program.

The first task performed by the main module is a preliminary check to ensure that, in the initial state of the domain, each passenger can be reached by at least one vehicle, and that the vehicle can then reach the passenger's destination. (Reachability also takes into account the amount of gas initially in the vehicle and the amount of gas needed to travel between locations.) This check is done by formulating a sub-problem $\Pi_2(AP, p)$ for each passenger p, so that $\Pi_2(AP, p)$ is consistent if-and-only-if p can be reached by some vehicle and then driven to his destination. The main module's task is then reduced to verifying whether all $\Pi_2(AP, p)$'s are consistent. The passenger that is to be considered is specified by an atom of the form $selected(p)$. The main rules of $\Pi_2(AP, p)$ are:

$$1\{\ assigned(P, V) : vehicle(V, M)\ \}1 \leftarrow selected(P).$$
$$\leftarrow \text{not } pass\_reachable\_from\_start.$$
$$\leftarrow \text{not } destination\_reachable\_from\_passenger.$$
$$pass\_reachable\_from\_start \leftarrow p\_location(S),\ reach\_from\_start(S, G).$$

The first rule states that exactly one vehicle should be assigned to drive the selected passenger. The two denials require that the assigned vehicle can reach the passenger from its initial location and can subsequently drive the passenger to his destination. As a result, $\Pi_2(AP, p) \cup \{selected(p)\}$ has an answer set if and only if passenger p can be reached by at least one vehicle that satisfies these requirements. The last rule defines reachability of the passenger in general terms of reachability of a location from the vehicle's initial location (with a certain amount of gas left at the end of the trip). Relation $destination\_reachable\_from\_passenger$ is defined in a similar way. Relation

$reach\_from\_start(s, g)$ is defined by the rules:

$reach\_from\_start(S, G) \leftarrow start(S),\ gas(G).$
$reach\_from\_start(Y, G - C) \leftarrow reach\_from\_start(X, G),\ connected(X, Y, C),\ G \geq C.$
$reach\_from\_start(X, T) \leftarrow reach\_from\_start(X, G),\ gasstation(X),\ tank(T).$
$start(S) \leftarrow assigned(P, V),\ vehicle\_at(V, S).$

The relation is formalized recursively. The first rule encodes the base case, and states that the start is reachable without using any gas. The next rule encodes the recursive step, and says that any location connected to the current location is reachable if enough fuel is left in the vehicle's tank; the amount of fuel in the tank at the end of the leg takes into account the cost of driving to the new location. The third rule considers the availability of a gas station and states that, if the current location is reachable from the start and has a gas station, then it is reachable from the start with a full tank left at the end of the trip. The last rule determines the start location of the vehicle currently assigned to the passenger; the rules for relations gas and $tank$ are similar.

If the preliminary test implemented by $\Pi_2(AP, p)$ succeeds, then $\Pi_1(AP)$ proceeds with the next phase of the search. In this phase, $\Pi_1(AP)$ maintains the current locations of passengers and vehicles and the gas level in the tank of each vehicle. The program selects one passenger p and assigns to him a vehicle v capable of taking him to his destination. The state of the domain is then updated according to the effects of driving p to his destination using v. Note that at this stage of the search we are only concerned with final locations of the objects and gas levels, and abstract from the low-level actions that need to be performed to drive p to his destination. At this point, the process repeats: $\Pi_1(AP)$ selects another passenger, assigns him a vehicle, and the search continues.

Whenever no vehicle can be found for driving a currently selected passenger, the search backtracks. To improve performance, the selection of passengers and vehicles is guided by a heuristic that prefers to use vehicles that are already at a passenger's current location. This is implemented by the rules:

$1\{\ use\_at\_passenger,\ \neg use\_at\_passenger\ \}1 \leftarrow \neg all\_at\_destination.$
$\leftarrow use\_at\_passenger, \text{not}\ some\_already\_at\_passenger.$
$\#minimize[\ use\_at\_passenger = 1,\ \neg use\_at\_passenger = 2\ ].$

The first rule states that if not all passengers are at their destinations, then it is possible to select between using vehicles that are at a passenger's location and vehicles that are not. The second rule states that it is impossible to require the use of a vehicle that is at a passenger's location if no vehicle is at this location. The last rule (from a language extension of CLASP) states that choosing to use vehicles that are not at a passenger's location has a penalty. The selection of a passenger and a vehicle is performed by the rules:

$$1\{\ assigned(P, V) : passenger(P) : \text{not}\ at\_destination(P) : vehicle(V, M) : good(V, P) : already\_at\_passenger(V, P)\ \}1 \leftarrow \neg all\_at\_destination,\ use\_at\_passenger.$$

$$1\{\ assigned(P, V) : passenger(P) : \text{not}\ at\_destination(P) : vehicle(V, M) : good(V, P)\ \}1 \leftarrow \neg all\_at\_destination,\ \neg use\_at\_passenger.$$

Both rules state that exactly one pair $\langle p, v \rangle$ must be selected. In the first rule, the selection is among the pairs for which p and v are at the same location. In the second rule,

this restriction is lifted. Next, $\Pi_1(AP)$ verifies the reachability of p from v's location (if necessary) and of p's destination after v has picked up p. The rules for the definition of reachability are the same as used in $\Pi_2(AP)$. Note that multiple paths may exist that allow v to drive p to his destination. For this reason, we consider only *best* paths, i.e. those that leave the largest amount of gas in v's tank at the end of the path. Note that if a solution to the main problem cannot be found by using best paths, then no solution can be found even if the condition is lifted. Considering explicitly multiple paths, in general, involves an amount of backtracking that would make performance unacceptable.

At this stage of the search, we focus on finding the amount of gas left that characterizes the best path. The amount is determined in two steps. First, relation $best\_d1\_gas(g)$ says that bg is the largest amount of gas left in v's tank after it has reached p's location:

$$best\_d1\_gas(BG) \leftarrow p\_location(D),\ BG = \#\max[\, reach\_from\_start(D, G) = G\,].$$

It should be noted that in the definition of $destination\_reachable\_from\_passenger$ used in $\Pi_2(AP, p)$, the amount determined by $best\_d1\_gas$ is used as the initial gas level for the trip to the passenger's goal location. We then define the similar relation $best\_dest\_gas(g)$:

$$best\_dest\_gas(BG) \leftarrow destination(D),\ BG = \#\max[\, reach\_from\_d1(D, G) = G\,].$$

The value g for which $best\_dest\_gas$ holds is the amount of gas left in v's tank after driving p to the airport along the best path.

Once $\Pi_1(AP)$ has determined a sequence of passenger-vehicle selections that successfully takes all passengers to their respective destinations, the sequence of actions to be performed for each passenger-vehicle pair is determined by means of another program, $\Pi_3(AP)$. The program $\Pi_3(AP)$ (i) takes as an input a pair $\langle p, v\rangle$ and the current state of the domain, and (ii) finds the sequence of actions corresponding to the best path for $\langle p, v\rangle$. The program is called iteratively for each passenger-vehicle assignment determined earlier by $\Pi_1(AP)$. Between calls, $\Pi_1(AP)$ updates the state of the domain according to the sequences generated by $\Pi_3(AP)$.

As in the Hydraulic-System-Planning benchmark, $\Pi_3(AP)$ is written in the language of ICLINGO, using the maximum length of the paths considered as the growth parameter. The search revolves around the notion of *extension of input graph G for vehicle v*: a directed graph whose nodes are pairs $\langle l, g\rangle$, where l is a location and g is an integer specifying an amount of gas. A pair $\langle l, g\rangle$ belongs to the extension E of G if l can be reached from the current location of v (in the current state of the domain) with an amount of gas g left in the tank. In $\Pi_3(AP)$, we consider paths in E of increasing length until we find the best path. The paths are represented by $arc(l, lg, n, ng, i)$, stating that the i^{th} element of a path is the arc from $\langle l, lg\rangle$ to $\langle n, ng\rangle$. The base of $\Pi_3(AP)$ is:

$$arc(S, SG, X, SG - C, 1) \leftarrow start(S),\ gas(SG),\ connected(S, X, C),\ SG \geq C.$$
$$arc(S, SG, S, T, 1) \leftarrow start(S),\ gas(SG),\ gasstation(SG),\ tank(T).$$

The rules define the first arc of each path in E, with the second rule dealing with the case in which the vehicle is refueled at the start. The cumulative part of $\Pi_3(AP)$ determines

the i^{th} arc in each path, where i is the growth parameter:

$$arc(X, G1, Y, G1 - C, i + 1) \leftarrow arc(Z, G0, X, G1, i),\ connected(X, Y, C),\ G1 \geq C.$$
$$arc(X, G1, X, T, i + 1) \leftarrow arc(Z, G0, X, G1, i),\ gasstation(X),\ tank(T).$$

The cumulative part also includes the definition of relation $at\_dest(i)$, saying that there exists a path of length i that leads v to the destination location (after picking up p) in such a way that the intended amount of gas is left in v's tank:

$$at\_dest(i) \leftarrow arc(X, G, D, BG, i),\ destination(D),\ best\_dest\_gas(BG).$$

Relation at_dest is the key to detecting when the best path has been found. Finally, the volatile part of $\Pi_3(AP)$ contains a denial $\leftarrow$ not $at\_dest(i)$. which intuitively forces the iterations to continue until the best path has been found. Once that occurs, the corresponding sequence of actions is generated by re-tracing the best path from its end, with the same approach used in the Hydraulic-System-Planning benchmark. By $\Pi(AP)$ we denote $\Pi_1(AP)$-$\Pi_3(AP)$.

7 Performance Assessment

In order to evaluate how well our tool selection and modeling techniques fared in the competition, we conducted a series of experiments on the competition instances (made publicly available after the end of ASPCOMP). All experiments were performed on a computer with an Intel i7 processor running at 3 GHz, 4 GB RAM and FedoraCore 11. The systems used were GRINGO 3.0.3, CLASP 1.3.7, ICLINGO 3.0.3 (with CLASP 1.3.5), BPROLOG 7.4 and EZCSP 1.6.20b33. Our goal was to compare the performance of our encodings with that of the pure ASP encodings made available by the ASPCOMP organizers[3] [1] and run using CLASP. Below, we label the pure ASP encodings by $\Pi_b(\cdot)$ (e.g. $\Pi_b(WA)$ is the pure ASP encoding for the Weight-Assignment benchmark). For Reverse-Folding benchmark no pure ASP encoding was available. We use $\Pi_2(RF)$ as the baseline. The timeout for each run was 600 seconds. The average times were computed by considering only the instances that did not time out.

The results (see Table 1) show that the encodings developed in this paper are substantially faster than the baseline encodings. In no case our encodings timed out, whereas the baseline encodings timed out a total of 22 times. The time taken by our encodings was between 1 and 3 orders of magnitude better than that of the baseline encodings, which is even more impressive considering that the instances that timed out were not used in computing the average times. We believe that the post-competition results clearly demonstrate the superior performance and scalability yielded by the encodings we developed. Detailed tables can be found on the EZCSP web page (http://marcy.cjb.net/ezcsp) together with the encodings described in this paper.

8 Conclusions

In this paper we have described our solutions to four challenging ASPCOMP problems. The solutions involved non-trivial use of solvers for CASP and IASP – selected out

[3] In these encodings we replaced all disjunctive rules by suitable choice rules.

	WA		RF		HP		AP	
	$\Pi_2(WA)$	$\Pi_b(WA)$	$\Pi_3(RF)$	$\Pi_2(RF)$	$\Pi'(HP)$	$\Pi_b(HP)$	$\Pi(AP)$	$\Pi_b(AP)$
Total	3.49	2158.44	88.61	9000.00	2.07	47.25	302.71	7077.21
T/O	0	0	0	15	0	0	0	7
Avg	0.23	143.90	5.91	–	0.16	3.63	20.18	359.65

Table 1. Performance comparison (T/O stands for number of timeouts).

of concerns for the scalability of the pure ASP solutions. Currently no programming methodology exists for these tools. We hope that our description has provided an outline of the methodology we followed and that this, albeit being expressed at this point in problem-specific terms, may constitute a first step in the development of a general methodology for the use of such advanced ASP solvers.

Acknowledgments. The idea to use irrelevant moves in the Reverse-Folding benchmark is by Selim Erdogan, who also gave valuable suggestions on this paper and was a member of the EZCSP team at ASPCOMP. Yuliya Lierler was supported by a CRA/NSF 2010 Computing Innovation Fellowship.

References

1. Third answer set programming competition (2011), https://www.mat.unical.it/aspcomp2011/
2. Balduccini, M.: A General Method To Solve Complex Problems By Combining Multiple Answer Set Programs. In: ICLP09 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP09) (Jul 2009)
3. Balduccini, M.: Representing Constraint Satisfaction Problems in Answer Set Programming. In: ICLP09 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP09) (Jul 2009)
4. Balduccini, M., Lierler, Y.: ASP-Based Problem Solving with Cutting-Edge Tools. In: ICLP11 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP11). pp. 14–28 (Jul 2011)
5. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: Engineering an Incremental ASP Solver. In: ICLP. pp. 190–205 (2008)
6. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-Driven Answer Set Solving. In: Veloso, M.M. (ed.) Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07). pp. 386–392 (2007)
7. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing 9, 365–385 (1991)
8. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic 7(3), 499–562 (2006)