University of Nebraska Omaha

University of Nebraska at Omaha
DigitalCommons@UNO

Computer Science Faculty Proceedings & Presentations

Department of Computer Science

2011

Weighted-Sequence Problem: ASP vs CASP and Declarative vs Problem-Oriented Solving

Yuliya Lierler
University of Nebraska at Omaha, ylierler@unomaha.edu

Shaden Smith
University of Kentucky

Miroslaw Truszczyński
University of Kentucky

Alex Westlund
University of Kentucky

Follow this and additional works at: https://digitalcommons.unomaha.edu/compsicfacproc

Part of the Computer Sciences Commons

Recommended Citation

Lierler, Yuliya; Smith, Shaden; Truszczyński, Miroslaw; and Westlund, Alex, "Weighted-Sequence Problem: ASP vs CASP and Declarative vs Problem-Oriented Solving" (2011). *Computer Science Faculty Proceedings & Presentations*. 32.
https://digitalcommons.unomaha.edu/compsicfacproc/32

This Conference Proceeding is brought to you for free and open access by the Department of Computer Science at DigitalCommons@UNO. It has been accepted for inclusion in Computer Science Faculty Proceedings & Presentations by an authorized administrator of DigitalCommons@UNO. For more information, please contact unodigitalcommons@unomaha.edu.

Weighted-Sequence Problem: ASP vs CASP and Declarative vs Problem-Oriented Solving

Yuliya Lierler, Shaden Smith, Miroslaw Truszczynski, Alex Westlund

Department of Computer Science, University of Kentucky, Lexington, KY 40506-0633, USA

Abstract. Search problems with large variable domains pose a challenge to current answer-set programming (ASP) systems as large variable domains make grounding take a long time, and lead to large ground theories that may make solving infeasible. To circumvent the "grounding bottleneck" researchers proposed to integrate constraint solving techniques with ASP in an approach called *constraint* ASP (CASP). In the paper, we evaluate an ASP system CLINGO and a CASP system CLINGCON on a handcrafted problem involving large integer domains that is patterned after the database task of determining the optimal join order. We find that search methods used by CLINGO are superior to those used by CLINGCON, yet the latter system, not hampered by grounding, scales up better. The paper provides evidence that gains in solver technology can be obtained by further research on integrating ASP and CSP technologies.

1 Introduction

ASP [9, 11] is a declarative programming formalism based on the answer-set semantics of logic programs [6]. It is oriented towards combinatorial search problems. Search problems with large variable domains pose a major challenge to the current generation of answer-set programming (ASP) systems, which require that the answer-set program representing the problem first be grounded by an ASP *grounder* and only then solved by an ASP *solver* [2]. The difficulty is that large variable domains make grounding take long, sometimes prohibitively long, time and result in large ground theories that often make solving infeasible, even though the problem may in fact be quite easy. Typical examples of problems with variables ranging over large domains are optimization problems, which require variables to represent possible values of goal function, and planning and scheduling problems that require variables to represent times when events can take place.

Constraint ASP [10, 5, 1] (CASP) integrates ASP with tools and techniques developed for constraint satisfaction problems (CSP). The goal of CASP systems is to address the grounding bottleneck of ASP. CASP solvers address the problem by performing partial grounding only, not grounding variables whose values range over large domains, but delegating the task of finding appropriate values for them to specialized algorithms such as constraint solvers.

In the work we report here we experimentally evaluated ASP and CASP systems. For our study we selected the highly optimized ASP system CLINGO[1] [4] that is based on the ASP grounder GRINGO [3] and the ASP solver CLASP [4], and a CASP system CLINGCON[2] [5] that is based on modifications of GRINGO, CLASP, and the constraint solver GECODE[3]. To conduct the experiments we handcrafted a benchmark called a *weighted-sequence* problem. The key features of the problem are inspired by the important industrial problem of finding an optimal join order by cost-based query optimizers in database systems. When selecting and designing the problem, we were motivated by the fact that it involved variables with large domains of integers, which made it well suited for our study. We were also motivated by the practical relevance of that problem and its hardness. Current query optimizers attempt to find an optimal join order only for joins consisting of relatively few tables (five tables in the case of the ORACLE optimizer [7, Page 416]). We modified the problem by introducing additional complexities to enrich its structure and create possibilities for non-trivial modeling enhancements requiring a deeper understanding of problem properties.[4]

In our experiments we aimed to understand relative advantages of sophisticated search procedures involving conflict-driven clause learning and backjumping of the ASP solver CLINGO versus the idea of limiting grounding and delegating some constraint solving tasks to a specialized constraint solver employed by CLINGCON – an idea central to CASP. We experimented with two sets of instances: a SMALL set of 30 instances, where the integer parameters were quite small, and a LARGE set also of 30 instances, where the integer parameters were substantially larger. Our key findings are that: the effectiveness of the search procedure used by CLINGCON lags behind that of CLINGO; and that circumventing the grounding bottleneck makes CLINGCON scale up substantially better.

The former finding is demonstrated by the running times we observed on instances in the SMALL set, where the integer parameters are low and grounding is not a major factor. On these instances, CLINGO in general performed better. Further evidence in support of that claim came from experiments with several encodings of the weighted-sequence problem, one of which represented the problem requirements literally as they appeared in the problem statement, while others also included constraints not given explicitly but derived from those stated directly. CLINGO was much less sensitive to modeling enhancements, suggesting that its learning techniques could infer at least some of the derived constraints. However, including these "derived" constraints had a major positive effect on CLINGCON, suggesting its search methods are not yet powerful enough to infer useful constraints when they are not given explicitly.

---

[1] `http://potassco.sourceforge.net/`

[2] `http://www.cs.uni-potsdam.de/clingcon/`

[3] `http://www.gecode.org/`

[4] The benchmark was submitted to and used in the Third Answer Set Programming Competition (`https://www.mat.unical.it/aspcomp2011/OfficialProblemSuite`). It was referred to as benchmark number 28, *Weight-Assignment Tree*.

The latter key finding concerning the scalability was evidenced by the results concerning the LARGE set of instances showing that when the parameters get larger, large sizes of grounded programs slow down CLINGO dramatically, while even less sophisticated search methods of CLINGCON are capable to find solutions quickly.

Our results strongly suggest the validity of the CASP approach but also point out that there is still much room for improvement in the way CASP systems do learning.

2 Problem Statement

In the weighted-sequence problem we are given a set of leaves (nodes) and an integer m — *maximum cost*. Each leaf is a pair $(weight, cardinality)$ where *weight* and *cardinality* are integers. Every sequence (permutation) of leaves is such that all leaves but the first are assigned a color. A colored sequence is associated with the *cost*. The task is to find a colored sequence with the cost at most m.

For a set S of leaves and an integer m, we denote the corresponding weighted-sequence problem by $[S, m]$. We say that an integer m is *optimal* with respect to a set S of leaves if m is the least integer u such that the weighted-sequence problem $[S, u]$ has a solution.

Let M be a sequence of n leaves $l_0, \ldots, l_{n-1}$. For each leaf l_i, $0 \leq i \leq n-1$, by $w(l_i)$ and $c(l_i)$ we denote its weight and cardinality, respectively. We color each leaf l_i, $1 \leq i \leq n-1$, *green, red*, or *blue*; the leaf l_0 is not colored. We define the costs of leaves as follows. For the leaf l_0, we set

$$cost(l_0) = w(l_0).$$

For every colored leaf l_i, $1 \leq i \leq n-1$, we set

$$cost(l_i) = \begin{cases} w(l_i) + c(l_i) & \text{if } l_i \text{ is green} \\ cost(l_{i-1}) + w(l_i) & \text{if } l_i \text{ is red} \\ cost(l_{i-1}) + c(l_i) & \text{if } l_i \text{ is blue.} \end{cases}$$

The cost of the sequence M is the sum of the costs of its colored leaves:

$$cost(M) = cost(l_1) + \cdots + cost(l_{n-1}).$$

3 ASP: Generate and Test Methodology

Answer set programming [9, 11] is a declarative programming formalism based on the answer set semantics of logic programs [6]. The idea of ASP is to represent a given computational problem by a program whose answer sets correspond to solutions. A common methodology to solve a problem in ASP is to design two main parts of a program: GENERATE and TEST [8]. The former defines a larger collection of answer sets that could be seen as potential solutions. The

latter consists of rules that eliminate the answer sets that do not correspond to solutions. Often a third part of the program, DEFINE, is also necessary to express auxiliary concepts that are used to encode the conditions of GENERATE and TEST. Thus, when we represent a problem in ASP, two kinds of rules have a special role: those that *generate* many answer sets corresponding to possible solutions, and those that can be used to *eliminate* the answer sets that do not correspond to solutions.

A typical logic programming *rule* has a form

$$a_0 \leftarrow a_1, \ldots, a_m,\ not\ a_{m+1}, \ldots,\ not\ a_n, \tag{1}$$

where each a_i $(0 \leq i \leq n)$ is an atom of the underlying language. We call the left-hand side (right-hand side) of the arrow symbol in a rule (1) the rule's *head* (*body*, respectively). Rules are used to describe relations between concepts represented by their atoms. Together, as a program, they specify a class of special models the program. These models are called *answer sets*. Informally speaking, answer sets are those models of a program that are in some very precise way "justified" by the program. We refer for the formal definition to the overview by Brewka et al. [2].

For instance, the program

$$p.$$
$$q \leftarrow p,\ not\ r.$$

is composed of two rules. The first rule is often called a *fact* since its body is empty and it represents the fact p. The second rule justifies the derivation of q, as we have p and the program has no way to justify r (no rule has r as its head). Consequently, by a form of the closed-world assumption, *not* r is true and the rule "fires." In this case, $\{p, q\}$ is the only model "justified" by the program, that is, the only answer set, even though the program has additional models.

In addition to rules of the form (1), GRINGO also accepts rules of other kinds. Two important examples are *choice rules* and *constraints*. For example, the rule

$$\{p, q, r\}.$$

is a choice rule (in this case, with the empty body). Informally, it justifies any (even empty) subset of $\{p, q, r\}$. Thus, any subset of $\{p, q, r\}$ is an answer set the program consisting of this rule only. As this example demonstrates, choice rules generate sets of models and are typically members of the GENERATE part of the program.

Constraints often form the TEST section of a program. Syntactically, a *constraint* is the rule with an empty head. It encodes the constraints of the problem that answer sets must have. For instance, the constraint

$$\leftarrow p,\ not\ q.$$

eliminates answer sets that include p and do not include q. When this constraint and the constraint $\leftarrow r$, which eliminates answer sets containing r, are conjoined

with the choice rule above, the resulting program has three answer sets: $\emptyset$, $\{q\}$ and $\{p, q\}$.

The input language of CLINGO (CLINGCON) allows the user to specify large programs in a compact fashion, using rules with schematic variables and other abbreviations. We refer the reader to the manual of CLINGO [3] for more details.

When processing, programs are first grounded by a grounder (a program like GRINGO). Afterwards, a solver program (for instance, CLASP; these programs share much similarity with propositional SAT solvers) searches for the answer sets of the propositional output of the grounding phase.[5] The problem is that the output of the grounder may be large. By exploiting constraint (CSP) solvers for some search tasks, one can get by with a smaller grounding. This is the idea behind CASP, which we compare here experimentally to the standard ASP solving method.

4 Encodings

ASP encodings of the weighted-sequence problem represent it as a logic program so that answer sets of the program correspond to sequences of leaves with the cost less than or equal to the given bound. Below we present several encodings. One of them simply represents literally the requirements as they appear in the problem statement. The remaining ones expand it by imposing additional constraints derived by analyzing the problem statement. All these encodings can be systematically transformed into the corresponding CLINGCON programs that take advantage of a special feature of CLINGCON, *constraint atoms*. We use the resulting CLINGCON programs in our experiments with CLINGCON.

There are several concepts that are common to all the encodings. Let n and m be integers giving the number of leaves in a weighted-sequence and the bound on the total cost of a solution (maximum cost), respectively. Then each encoding contains the following facts

$$num(n)$$
$$maxCost(m).$$

The weight and cardinality of each leaf is specified by facts of the form

$$leafWeightCard(i, w, c)$$

where i is an integer that ranges from 1 to n and stands for an *id* of a leaf, and w and c are the weight and cardinality of this leaf, respectively.

In addition, the DEFINE part of every encoding presented here contains the rules

$$position(X) \leftarrow X = 0..N-1, num(N)$$
$$coloredPos(X) \leftarrow X = 1..N-1, num(N),$$

which specify that there are n positions $0 \ldots n-1$ in the sequence, and that the positions $1 \ldots n-1$ are colored and the position 0 is not.

[5] CLINGO, the program we study in this paper, simply combines the two programs into one.

Declarative Encoding: The GENERATE part of a *declarative* encoding, DECL, consists of two components. The first one generates a sequence by assigning each leaf its position. It is formed by the following two rules:

$$1\{\mathit{leafPos}(L,P) : \mathit{position}(P)\}1 \leftarrow \mathit{leaf}(L)$$
$$1\{\mathit{leafPos}(L,P) : \mathit{leaf}(L)\}1 \leftarrow \mathit{position}(P).$$

Intuitively, the first rule says that each leaf is assigned exactly one position. The second rule ensures that each position holds exactly one leaf.

The second component of the GENERATE part assigns exactly one color to every colored position in a sequence (positions $1, \ldots, n-1$). To this end, it uses the rule

$$1\{\mathit{posColor}(P,C) : \mathit{color}(C)\}1 \leftarrow \mathit{coloredPos}(P). \tag{2}$$

The DEFINE part of the program DECL includes the rules that specify the cost of each colored leaf in a sequence. For instance, the two rules

$$\begin{array}{l} \mathit{posCost}(0,\mathit{Cost}) \leftarrow \mathit{leafWeightCard}(L,\mathit{Cost},C), \mathit{leafPos}(L,0) \\ \mathit{posCost}(P,\mathit{Cost}) \leftarrow \mathit{coloredPos}(P),\ \mathit{posColor}(P,\mathit{red}),\ \mathit{leafPos}(L,P), \\ \qquad\qquad \mathit{leafWeightCard}(L,W,C),\ \mathit{posCost}(P-1,\mathit{Cost}'), \\ \qquad\qquad \mathit{Cost} = \mathit{Cost}' + W \end{array} \tag{3}$$

state that (i) the cost of the leaf in position 0 is its weight, and (ii) the cost of the leaf in position P that is colored *red* is the sum of its weight and the cost of the preceding node. Similar rules specify costs of leaves when they are colored *green* or *blue*. The DEFINE part of DECL also contains rules that define the cost of a sequence:

$$\begin{array}{l} \mathit{seqCost}(1,\mathit{Cost}) \leftarrow \mathit{posCost}(1,\mathit{Cost}) \\ \mathit{seqCost}(P,\mathit{Cost}) \leftarrow \mathit{coloredPos}(P),\ P > 1,\ \mathit{seqCost}(P-1,C), \\ \qquad\qquad\qquad\qquad \mathit{posCost}(P,C'),\ \mathit{Cost} = C + C'. \end{array} \tag{4}$$

Consequently, an answer set contains the ground atom $\mathit{seqCost}(n-1,c)$ if and only if c is the number that corresponds to the cost of the sequence determined by other ground atoms in this answer set (we recall that n is the number of leaves).

Finally, DEFINE includes the rule that introduces an auxiliary predicate *exists*:

$$\mathit{exists} \leftarrow \mathit{seqCost}(N-1,\mathit{Cost}),\ \mathit{num}(N),\ \mathit{Cost} \leq M, \mathit{maxCost}(M) \tag{5}$$

which affirms that the sequence determined by an answer set has total cost within the specified bound m.

The TEST part of DECL contains a single constraint:

$$\leftarrow \mathit{not}\ \mathit{exists}$$

It tests whether an answer set contains the atom *exists* and eliminates those that do not. In this way only answer sets determining sequences with the total cost

within the specified bound remain. If no such sequence exists the program has no answer sets.

We note that the rules in (3) and (4) may be augmented by additional conditions in the bodies

$$Cost \leq M,\ maxCost(M).$$

This modification is crucial for making grounded instances of programs smaller and is incorporated in our encodings.

Sequence Encoding: For a leaf l, we define its *value* $val(l)$ as the smaller of the two numbers, the weight and the cardinality, associated with l. That is,

$$val(l) = \begin{cases} w(l) & \text{if } w(l) \leq c(l), \\ c(l) & \text{otherwise.} \end{cases}$$

Let l and l' be two leaves in a sequence so that l immediately precedes l'. We define the *color number* of the leaf l' to be

$$colorNum(l') = min(w(l') + c(l'), cost(l) + val(l')).$$

Let us assign a color to every leaf l' in a colored position according to the formula:

$$color(l') = \begin{cases} green & \text{if } colorNum(l') = w(l') + c(l') \\ red & \text{otherwise, if } colorNum(l') = cost(l) + w(l') \\ blue & \text{otherwise, if } colorNum(l') = cost(l) + c(l') \end{cases}$$

where l precedes l' in the sequence.

Observation 1: Any color assignment different from the one defined above results in a colored sequence with the same or higher cost.

Observation 1 represents a property of the weighted-sequence problem that is not explicitly present in the problem statement and so, it is not a part of the DECL encoding. It is the basis for the *sequence* encoding SEQ that builds upon the DECL encoding by replacing the "non-deterministic" color-choice rule (2) with a set of "deterministic rules." For instance,

$$\begin{aligned} posColor(P, green) \leftarrow\ & P > 1,\ coloredPos(P),\ leafPos(L, P), \\ & leafWeightCard(L, W, C),\ leafValue(L, V), \\ & posCost(P - 1, Cost),\ W + C < Cost + V \end{aligned}$$

is one of the rules in this set (for a leaf l, *leafValue*(l, v) is defined to hold precisely when $v = val(l)$).

Intuitively, the advantage of the encoding SEQ over DECL is a reduced search space as color assignment requires no choices. However, by Observation 1, no optimal solutions are lost while some suboptimal ones are pruned. We note that an additional (minor) simplification results from the fact that in the DECL encoding, three cases are considered when a position is colored *green*, *red*, and *blue*. In the encoding SEQ, with the use of *leafValue* predicate, it is sufficient to consider two cases only: when position is colored green and when it is not.

Sequence Encoding+:
Observation 2: Let l and l' be two consecutive elements in a sequence M (in that order), neither being a green-colored leaf. It is easy to see that if $val(l') < val(l)$ then the sequence M' constructed from M by changing the order of l and l' has a smaller cost than M, i.e., $cost(M') < cost(M)$.

Observation 2 allows us to add the constraint

$$\begin{array}{l} \leftarrow \mathit{coloredPos}(P; P-1), \\ \quad \mathit{not}\ \mathit{posColor}(P, \mathit{green}),\ \mathit{not}\ \mathit{posColor}(P-1, \mathit{green}), \\ \quad \mathit{leafPos}(L, P-1),\ \mathit{leafPos}(L', P), \\ \quad \mathit{leafValue}(L, V),\ \mathit{leafValue}(L', V'), V > V'. \end{array} \tag{6}$$

to the SEQ encoding. We denote the resulting program by SEQ+.

The idea behind extending the SEQ encoding with (6) is that it reduces the search space. Observation 2 implies that no optimal solutions to the weighted-sequence problem are lost because of the additional constraint, some suboptimal ones will in general be pruned.

Sequence Encoding++:
Let $g_1, \ldots, g_k$ be a set of all green nodes in a sequence M, that is,

$$M = M_0\, g_1\, M_1 \ldots g_k\, M_k \tag{7}$$

where each M_i, $0 \leq i \leq k$, is a sequence of non-green leaves. We call M_0 the $0th$ *partition* of (7) and each $g_i\, M_i$, $1 \leq i \leq k$, a *green partition* of (7).

Observation 3: The fact that the cost of a green node only relies on its own weight and cardinality makes it evident that the cost of the sequence (7) is the same as the cost of the sequence $M_0\ P$, where P is any permutation of the set of green partitions of (7), $\{g_1\, M_1, \ldots, g_k\, M_k\}$.

Observation 3 allows us to add a constraint

$$\begin{array}{l} \leftarrow \mathit{leafPos}(L, P),\ \mathit{leafPos}(L', P'), \\ \mathit{posColor}(P, \mathit{green}),\ \mathit{posColor}(P', \mathit{green}), \\ L < L',\ P > P' \end{array}$$

to the SEQ+ encoding. We denote the resulting program by SEQ++. Intuitively, the last rule "breaks the symmetry" by enforcing that any answer set to the program has the green leaves in the corresponding solution sequence sorted according to their costs.

Clingcon Encodings: The CASP language of CLINGCON extends the ASP language of CLINGO by introducing "constraint atoms". These atoms are interpreted differently than "typical" ASP atoms. The system CLINGCON splits the task of search between two programs: an ASP solver (CLASP) and a CSP solver (GECODE). The ASP solver incorporated in CLINGCON treats constraint atoms as boolean atoms and assigns them some *truth* value. The CSP solver, on the other hand, is used to verify whether the assignments given to the constraint atoms by the ASP solver of CLINGCON hold based on their "real" meaning.

Let us note that *posCost* and *seqCost* predicates used in all CLINGO encodings are "functional". In other words, when this predicate occurs in an answer set its first argument uniquely determines its second argument. Often, functional predicates in ASP encodings can be replaced by constraint atoms in CASP encodings. Indeed, this is the case in the weighted-sequence problem domain. This allows us to create alternative encodings for DECL, SEQ and the extensions of SEQ.

We note that only the rules containing functional predicates *posCost* and *seqCost* were changed in DECL and SEQ and its extensions to produce CLINGCON programs. For instance, the rules in (4) and (5) have the following form in the CLINGCON encodings

$$\begin{array}{l} \mathit{seqCost}(1) =^{\$} \mathit{posCost}(1) \ \leftarrow \ \mathit{coloredPos}(1) \\ \mathit{seqCost}(P) =^{\$} \mathit{posCost}(P) + \mathit{seqCost}(P-1) \ \leftarrow \ P > 1, \ \mathit{coloredPos}(P) \\ \mathit{exists} \ \leftarrow \ \mathit{seqCost}(N-1) \leq^{\$} M, \ \mathit{num}(N), \ \mathit{maxCost}(M), \end{array}$$

where

$$\mathit{seqCost}(1) =^{\$} \mathit{leafCost}(1), \ \mathit{seqCost}(P) =^{\$} \mathit{leafCost}(P), \ \mathit{seqCost}(N-1) \leq^{\$} M$$

are constraint atoms. The rules defining *posCost*, such as (3), are rewritten in a similar manner:

$$\begin{array}{l} \mathit{posCost}(0) =^{\$} \mathit{Cost} \leftarrow \mathit{leafWeightCard}(L, \mathit{Cost}, C), \mathit{leafPos}(L, 0) \\ \mathit{posCost}(P) =^{\$} \mathit{posCost}(P-1) + W \leftarrow \mathit{coloredPos}(P), \ \mathit{posColor}(P, \mathit{red}), \\ \qquad\qquad\qquad\qquad\qquad\qquad \mathit{leafPos}(L, P), \ \mathit{lwc}(L, W, C) \end{array}$$

where $\mathit{posCost}(0) =^{\$} \mathit{Cost}$ and $\mathit{posCost}(P) =^{\$} \mathit{posCost}(P-1)+W$ are constraint atoms.

We may benefit from the CLINGCON encodings when weights, cardinalities, and maximum cost of a given weighted-sequence problem are "large" integers. In such cases, any CLINGO encoding (that we were able to come up with) faces the *grounding bottleneck*. The size of the grounded CLINGO program heavily depends on the integer values provided by the problem specification. On the other hand, the size of the corresponding grounded CLINGCON program is only affected by these integer values to a small degree or, even, not affected at all.

5 Experimental Analysis

We first describe hardware specifications, the instance generation method, and the procedures used to perform all experiments. Then we discuss the experimental results reported.

Experiments were performed concurrently on several identical machines, each with a single-core 3.60GHz Pentium 4 CPU and 3Gb of RAM, and running Ubuntu Linux version 10.04. Experiments were performed with CLINGO version 3.0.3 and CLINGCON version 0.1.2.

Instance generation is driven by two inputs: the number of leaves in the instance, n, and the maximum value of a weight and cardinality of a single leaf, v. First, the set S of n leaves is created by generating random weights $w_0, \ldots, w_{n-1}$ and cardinalities $c_0, \ldots, c_{n-1}$ so that $0 \leq w_i, c_i \leq v$. For all instances in SMALL we used $v = 12$ and $n = 10$. For all instances in LARGE we used $v = 100$ and $n = 8$.

As leaves are created they are assigned a unique position in a sequence M. Positions 1 through $n-1$ in M are then randomly assigned colors *green*, *red*, or *blue*. We calculate the total cost m of the resulting colored sequence M and use it, together with S, as an instance to the weighted-sequence problem, denoted by $[S, m]$.

Thirty random problem instances generated in the way described above form the first set of instances, called *easy*, in the SMALL and in the LARGE sets, respectively. Clearly, all of these instances are satisfiable.

To create harder instances required an encoding and a solver. We chose the encoding SEQ++ along with CLINGCON. We proceeded by starting with an instance $[S, m]$ in the set of easy instances. We used CLINGCON to solve it, and if the instance was satisfiable, we calculated the tree cost for the solution found, $\hat{m}$, (clearly, $\hat{m} \leq m$). We then repeated the process for the instance $[S, \hat{m}-1]$. When $[S, \hat{m}-1]$ was found unsatisfiable, it indicated that $\hat{m}$ was optimal with respect to S, and $\hat{m}-1$ made the set S "barely" unsatisfiable (to be more precise, we used a version of binary search here to speed the process up). Instances obtained in this way from the *easy* instances formed the sets of *optimal* and *unsatisfiable* instances, respectively. The instances $[S, \hat{m}+5]$ formed the set of *hard* instances in the SMALL SET, and the instances $[S, \hat{m}+50]$ formed the set of *hard* instances in the LARGE SET. As before, we constructed groups of thirty hard, optimal and unsatisfiable instances for both SMALL and LARGE sets.

We used each instance with all encodings we considered. A time limit of 1500 seconds (25:00 minutes) was enforced for each instance. From each solve the grounding time, solving time, solution, the number of choices and the sizes of ground theories were recorded for further study.

We now present and discuss the results of our experiments. Due to space limits only summary results are presented here. For the encodings we used and the complete results, we refer to `http://www.csr.uky.edu/WeightedSequence/`. The first two tables, Table 1 and Table 2, concern SMALL and LARGE sets of instances, respectively. In the case of each set we considered its easy, hard, optimal and unsat subsets, and for each subset included a row in the table. There are two groups of columns in each table, one for CLINGO and the other one for CLINGCON. The columns in each group represent encodings DECL, SEQ, SEQ+ and SEQ++. Each entry in the table contains either the average running time of CLINGO or CLINGCON, respectively, for the set of 30 instances in the corresponding easy, hard optimal and unsat subset. However, if for at least one instance in the group we had a timeout, instead of the average running time we report the number of timeouts in the group.

Table 1. SMALL Instances

	CLINGO				CLINGCON			
Instance	DECL	SEQ	SEQ+	SEQ++	DECL	SEQ	SEQ+	SEQ++
Easy	0.88	0.75	0.81	0.86	0.02	0.06	0.05	0.15
Hard	4.01	1.19	1.77	2.97	**to=7**	9.50	4.34	5.04
Optimal	26.28	15.75	20.41	15.04	**to=27**	253.30	203.75	34.57
Unsat	180.62	193.79	162.88	27.88	**to=30**	**to=25**	**to=17**	128.63

Table 2. LARGE Instances

	CLINGO				CLINGCON			
Instance	DECL	SEQ	SEQ+	SEQ++	DECL	SEQ	SEQ+	SEQ++
Easy	21.76	15.38	15.35	16.73	0.01	0.07	0.07	0.08
Hard	22.75	13.96	14.41	23.36	**to=4**	1.76	1.18	0.72
Optimal	**to=1**	97.38	**to=1**	101.95	**to=24**	46.58	23.49	5.07
Unsat	**to=12**	**to=12**	**to=10**	248.81	**to=30**	189.38	92.29	10.43

Before we discuss the results as they pertain to comparisons of CLINGO versus CLINGCON, and to the role of explicit modeling of additional domain knowledge, we note that both tables show the increasing hardness of our instances as we move from easy to hard to optimal and, finally, to unsat ones. That is, the tables support the soundness of our approach to generate increasingly harder instances by lowering the bound for the total weight.

Effectiveness of search. We consider first the SMALL set of instances. When run on easy instances in that group, CLINGCON outperforms CLINGO. However, already on hard instances, the situation reverses. CLINGCON running times are worse and it times out on seven instances under the encoding DECL. The trend continues when we move on to optimal and unsat instances — CLINGCON performance deteriorates. The results suggest that for problems in that group, due to relatively small integer parameters used, neither the time needed for the compete grounding nor the size of the ground theory seem to have much negative effect on CLINGO, whose efficient and highly optimized search techniques more than compensate for that. On the other hand, worse (and, in the case of optimal and unsat instances, significantly worse) performance of CLINGCON suggests its search techniques lag behind those of CLINGO.

Our results provide also support to the claim of the importance of constraint learning while solving. CLINGO exploits sophisticated conflict-driven clause learning algorithm. The encodings we considered differ in that they represent progressively more and more problem constraints. The encoding used has very limited effect on the performance of CLINGO, when it is run on instances from the SMALL set. The only exception comes from the SEQ++ encoding resulting in a much better performance of CLINGO when run on the group of unsat instances. On

the other hand, the choice of the encoding has a major effect on CLINGCON. The results concerning instances in the LARGE set (Table 2), show a similar behavior. The effect of extra domain knowledge on the performance of CLINGO (while no longer negligible) is still much smaller that the effect it has on CLINGCON.

Thus, it seems that search techniques of CLINGO can learn some or even a major portion of the missing constraints, while CLINGCON search methods are not effective enough in that respect, as they benefit greatly when the constraints are provided explicitly.

Next, we note that instances in SMALL set where generated for $n = 10$ leaves (tables in the join problem), while in the LARGE set for $n = 8$. The problems in the LARGE set turn out to be easier for CLINGCON than those in the SMALL set. This suggests that CLINGCON handles increasing weights well but is more sensitive to changes in other parameters (line n). This observation is yet another indication of CLINGCON's weaker search techniques on the ASP side.

Lastly, we note briefly that while the additional constraints modeled explicitly in SEQ and SEQ+ encodings do help CLINGCON, it is the symmetry-breaking constraint used in SEQ++ that is particularly beneficial. In fact, it also seems to have a significant positive effect on CLINGO, as evidenced by the performance of CLINGO on unsat instances in the set LARGE (cf. Table 2).

In summary, our discussion above shows that there seem to be much room for improvement as concerns overall performance of search in CASP tools such as CLINGCON. It shows that sophisticated search techniques can compensate for some of the "derived" constraints not explicitly present in the problem statement, but also that some types of constraints, such as symmetry-breaking, make a difference even if solvers use a sophisticated constraint-learning algorithms.

Scalability: The results from Tables 3 and 4 provide evidence that CLINGCON scales up better than CLINGO as weights go up. To argue that we recall (cf. Table 1) that for the instances in the SMALL set, CLINGCON performs worse than CLINGO (except for instances that are easy). However when we move to instances in the LARGE set, the situation changes (cf. Table 2). Except for the encoding DECL, where it timed out frequently and much more often than CLINGO (as we argued above, due to its inability to learn useful constraints), CLINGCON completed the computation for every instance under SEQ, SEQ+ and SEQ++ encodings. CLINGO, on the other hand, times out on 23 instances under these three encodings (12 unsat instances under SEQ, one optimal and 10 unsat under SEQ+) and when it does not time out, its running times are much worse than those of CLINGO (one order of magnitude difference for the encoding SEQ++.

These results suggest that CLINGCON successfully addresses the grounding bottleneck resulting from large integer domains. To see this let us consider the sizes of ground theories (measured as the numbers of clauses and reported as averages over 30 instances that form each group).

First, we note that the sizes of CLINGCON encoding do not vary as we move from easy down to unsat instances. It is because these CLINGCON instantiates only non-weight, non-cardinality variables (such as the number of leaves and the number of colored positions) and they do not change. The only parameter that

Table 3. Sizes of ground programs: SMALL Instances

	CLINGO				CLINGCON			
Instance	DECL	SEQ	SEQ+	SEQ++	DECL	SEQ	SEQ+	SEQ++
Easy	75268	62739	63099	64719	575	539	899	2519
Hard	29326	26588	26948	28568	575	539	899	2519
Optimal	26842	24495	24855	26475	575	539	899	2519
Unsat	26350	24077	24437	26057	575	539	899	2519

Table 4. Sizes of ground programs: LARGE Instances

	CLINGO				CLINGCON			
Instance	DECL	SEQ	SEQ+	SEQ++	DECL	SEQ	SEQ+	SEQ++
Easy	1546714	1162451	1162619	1163207	383	358	526	1114
Hard	434237	377120	377288	377876	383	358	526	1114
Optimal	350933	308383	308551	309139	383	358	526	1114
Unsat	349336	307052	307220	307808	383	358	526	1114

distinguishes between the encodings, the bound on the weight, m, does not affect CLINGCON grounding size as, unlike in the CLINGO encodings, no "groundable" variable in CLINGCON encodings ranges over the domain $[0..m]$.

This brings us to the second observation, directly relevant for our study. Grounded CLINGCON encodings are much smaller than those resulting from the CLINGO ones. For instances in the LARGE set, the ground programs considered by clingo are quite large (hundreds of thousands of ground rules) and when constraints become tight (for optimal and unsat problems), very hard for CLINGO to process successfully within the time bounds set.

Next, we note that, not surprisingly, it is not only the size of the grounded program that matters. Easy instances result, after grounding, in much larger programs than unsat ones but as the constrains are not tight, the search process can terminate quickly. It is the combination of a large size and tight constraints that slows CLINGO down. Tight constraints are clearly a problem for CLINGCON, too (cf. the tables reporting the running times). But since the size of the ground theory it has to deal with is low, it does not hamper its performance on the ASP side of search, and the types of constraint problems CLINGCON delegates to GECODE are handled well (at least for our instances) by that CSP solver.

6 Conclusions and Future Work

Our experimental findings suggest several observations. Highly tuned ASP search algorithms (specifically, CLINGO) display a similar behavior on both "literal" (DECL) and "sophisticated" (SEQ, SEQ+ and SEQ++) encodings of a weighted-

sequence problem (although, as the instances get larger, symmetry-breaking incorporated into SEQ++ seems to start showing a noticeable benefit). The sophisticated encodings impose a number of restrictions on the problem's search space in comparison with the literal encoding and, our results show, for hybrid systems such as CLINGCON (that combines both ASP and CSP techniques in its search), it is of importance. Reduced search space that results can have a significant positive effect on their performance. Thus, our results suggest that the effectiveness of the search procedure used by CLINGCON lags behind that of CLINGO. They show that the problem seems to be with lack of strong learning techniques in CLINGCON. Including extra domain knowledge explicitly into problem representations greatly improves CLINGCON performance.

In the same time, we observed that CLINGCON scales up better than CLINGO and we attribute it to the fact that search in CLINGCON has to process smaller search spaces due to limiting the scope of grounding. Thus, CASP promises to become a milestone in declarative problem solving by providing the means of solving ASP grounding bottleneck. However, based on our work, we believe that to reach its full potential, CASP search methods need to incorporate learning to a much larger degree than they do so now. Certainly, other factors may be of importance, too, such as th enhanced communication between ASP and CSP processes while solving. The effect of those factors still needs to be evaluated.

We also stress that when we claim better scalability of CLINGCON we have in mind scalability with the size of weights (and cardinalities) going up. When we increase the number of leaves (and keep weight small) we expect the picture most likely would be different simply because of stronger search methods implemented in CLINGO.

This research focused on a single problem. In order to study the degree to which our findings are generizable, in the future work we will consider additional problems with large integer parameters and subject them to a similar study. On the other hand, we will also consider in more depth the problem that inspired the benchmark we considered here, the optimal join order problem, and study the effectiveness of ASP/CASP/CSP tools in solving it. This work is already under way.

Acknowledgments

We are grateful to Philip Cannata for bringing the problem of finding an optimal join order in query optimization to our attention, to Vladimir Lifschitz and Yuanlin Zhang for useful discussions related to the topic of this work, and to Max Ostrowski for suggestions on CLINGCON encodings. Yuliya Lierler was supported by a CRA/NSF 2010 Computing Innovation Fellowship, Miroslaw Truszczynski by the NSF grant IIS-0913459, and Shaden Smith and Alex Westlund by the NSF REU Supplement to that grant.

References

1. Balduccini, M.: Representing constraint satisfaction problems in answer set programming. In: Proceedings of ICLP'09 Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'09) (2009)
2. Brewka, G., Eiter, T., Truszczynski, M.: Answer set programming at a glance. Communications of the ACM (2011), to appear in December 2011
3. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A User Guide to `gringo`, `clasp`, `clingo` and `iclingo` (2010), `http://cdnetworks-us-2.dl.sourceforge.net/project/potassco/potassco_guide/2010-10-04/guide.pdf`
4. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Proceedings of 20th International Joint Conference on Artificial Intelligence (IJCAI'07). pp. 386–392. MIT Press (2007)
5. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In: Proceedings of ICLP-2009. pp. 235–249 (2009)
6. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R., Bowen, K. (eds.) Proceedings of International Logic Programming Conference and Symposium. pp. 1070–1080. MIT Press (1988)
7. Lewis, J.: Cost-Based Oracle Fundamentals. Apress (2005)
8. Lifschitz, V.: Answer set programming and plan generation. Artif. Intell. 138(1-2), 39–54 (2002)
9. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: The Logic Programming Paradigm: a 25-Year Perspective, pp. 375–398. Springer Verlag (1999)
10. Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. Annals of Mathematics and Artificial Intelligence (2008)
11. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25, 241–273 (1999)