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. We define the *weighted-sequence* problem inspired by an important industrial problem in ORACLE query optimization. We evaluate several approaches to solving this problem using answer set programming (ASP) system CLINGO and constraint answer set programming (CASP) solver CLINGCON. The focus of this paper is an experimental evaluation of search procedures behind CLINGO and CLINGCON. We used instances of the weighted-sequence problem where integer parameters were quite small so that grounding was not an issue and we could focus on comparison of the effectiveness of the CLINGO and CLINGCON solvers. Our key finding is that the effectiveness of the search procedure used by CLINGCON lags behind that of CLINGO. It points to the need for further research on the integration of ASP and CSP technologies.

1 Introduction

In this paper we define a *weighted-sequence* problem and evaluate several approaches to solving this problem using answer set programming system CLINGO[1] [5] (based on grounder GRINGO [3] and answer set solver CLASP[2] [5]) and constraint answer set programming solver CLINGCON[3] [6] (based on modifications of grounder GRINGO, answer set solver CLASP, and constraint solver GECODE[4]).

The weighted-sequence problem is inspired by the problem of finding an optimal join order in the cost-based query optimizer of ORACLE. Lewis [8, Chapter 14] describes the basics behind the procedure used by the ORACLE query optimizer for computing the optimal join order. A *join order* considered by ORACLE is a left-deep tree [14] such that (i) tables participating in the join are its leaves and (ii) each inner node is associated with one of the three possible join tags. Each table is associated with parameters that are used to compute the cost of each inner node with respect to the join tag assigned to it. The *join-order cost* is the sum of the costs of its inner nodes. The *optimal* join order is an order

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

[2] CLASP is an answer set solver that implements conflict-driven learning and backjumping.

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

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

with the minimal cost. Currently, ORACLE optimizer *may* consider all possible orderings only for the joins consisting of at most five tables [8, Page 416]. Thus, for the joins of more than five tables ORACLE will not even attempt to find an optimal join order. The ultimate goal of our project is to address the problem of finding an optimal join order using the state-of-the-art technology especially designed for solving difficult combinatorial search problems. As a step in this direction we devised a combinatorial problem based on similar constraints as those that specify the join-order problem, simplifying and modifying some of them. We call the resulting problem the *weighted-sequence* problem. In this paper we study the effectiveness of "pure" and constraint answer set programming tools in solving the weighted-sequence problem, selecting CLINGO and CLINGCON as representative systems of each type. Based on our experimental results we formulate several observations concerning the comparison between the two systems and the methodology of modeling problems in (constraint) answer set programming. The weighted-sequence problem was one of the benchmarks used in the Third Answer Set Programming competition[5]. It was referred to as benchmark number 28, *Weight-Assignment Tree*.

(Pure) answer set programming [10, 12] is a declarative programming formalism based on the answer set semantics of logic programs [7] that is oriented towards difficult combinatorial search problems. Constraint answer set programming [11, 6, 1] is an approach to declarative problem solving that integrates answer set programming with constraint processing (CSP). It is thus closely related to satisfiability modulo *theory* formalisms [13] that integrate satisfiability solving with specialized solvers for formulas in the corresponding *theory*. One of the goals of research on constraint answer set programming (CASP) systems is to address the grounding bottleneck of answer set programming (ASP) technology. Combinatorial optimization problems when modeled as answer set programs require the use of variables representing possible values of the goal function, values that often form large numeric domains. For instance, planning and scheduling problems give rise in the corresponding answer set programs to variables representing times when events take place. Typically, the ranges of possible values for these variables are large. Large domains are responsible for the sizes of ground programs to be prohibitively large, and grounding in such situations is a computationally heavy task. The optimization problem may in fact be quite easy (especially if optimal solutions are not required and approximate solutions are regarded as satisfactory). However, if grounding involves instantiations of variables over large domains, we may time out even before we get to the solving stage. CASP solvers attempt to 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, in particular, to CSP solvers.

The weighted-sequence problem is a benchmark that involves large domains of integers that can serve as possible values of the parameters associated with nodes, costs of nodes and the total cost of the sequence. In this work we ex-

[5] `https://www.mat.unical.it/aspcomp2011/OfficialProblemSuite`

perimentally evaluate search procedures behind ASP system CLINGO and CASP system CLINGCON. In the phase of the work we describe in this preliminary report, we used instances of the weighted-sequence problem where integer parameters were quite small. Thus, grounding was not an issue. In this way we could focus on comparisons of the effectiveness of the CLINGO and CLINGCON solvers. Our key finding is that the effectiveness of the search procedure used by CLINGCON lags behind that of CLINGO. It points to the need for further research on the integration of ASP and CSP technologies. Clearly, even if CASP tools address the grounding bottleneck, without solid performance of constraint solvers and good communication between ASP and CSP sides, CASP will not become the milestone in declarative problem solving that it promises to be. In our current and forthcoming work we will construct instances of the weighted-sequence problem with large weights. We will use these instances to investigate the effectiveness of CLINGCON in processing them. We expect that grounding will not be a problem and will focus on the question how close to optimal solutions will CLINGCON be able to get.

Another goal of the present work is to evaluate different approaches to modeling the weighted-sequence problem. First, we model the weighted-sequence problem in a purely declarative manner by simply encoding the problems requirements directly and literally as they appear in the problem statement. We then analyze the structure of the problem, and derive additional constrains, not explicitly stated. This allows us to formulate alternative encodings of the problem. We evaluate these encodings experimentally and use the results to formulate some plausible explanations pertaining to properties of different encodings, as well as to properties of the CLINGO and CLINGCON solvers.

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 the leaves is such that all but one of the leaves are assigned a color. A colored sequence is associated with the *cost*. The task is to find a colored sequence with the cost of at most m.

For a set S of leaves and an integer m, by $[S, m]$ we denote the corresponding weighted-sequence problem. We say that an integer m is *optimal* with respect to a set S of leaves if (i) there exists a solution for a weighted-sequence problem $[S, m]$ and (ii) there exists no solution for $[S, m-1]$.

We now describe how a cost of a sequence is defined. Let M denote a sequence of n leaves $l_0, \ldots, l_{n-1}$. For a leaf l_i in M $(0 \leq i \leq n-1)$, by l_i^w and l_i^c we denote its weight and cardinality respectively. Each leaf from $l_1, \ldots, l_{n-1}$ in M is assigned one of the three colors *green*, *red*, or *blue* (l_0, that is, the leaf in the position 0 in M, is not assigned any color). We define the costs of leaves as follows. For the leaf l_0, we set

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

For every colored leaf we set

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

where $1 \leq i \leq n-1$. The cost of the sequence M is the sum of the costs of its colored leaves, i.e.,

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

3 ASP: Generate and Test Methodology

Answer set programming [10, 12] is a declarative programming formalism based on the answer set semantics of logic programs [7]. 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 [9]. 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 constraints. 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.

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. Also, such aggregates as *count* and *sum* [2] are accepted by CLINGO. We refer the reader to the manual of CLINGO [4] for more details.

4 Encodings

We start by presenting an encoding in the CLINGO language that attempts to solve the weighted-sequence problem in a purely *declarative* manner by simply encoding directly and literally the requirements as they appear in the problem statement. We then describe two alternative encodings of the problem, referred to as *sequence* and *partition*, that impose additional constraints derived by analyzing the problem statement. Finally, we discuss how the presented CLINGO encodings may be mapped into the corresponding CLINGCON programs that take advantage of a special feature of CLINGCON, *constraint atoms*. In the subsequent section, we evaluate these encodings experimentally and discuss the results of the experiments.

In an ASP approach to the weighted-sequence problem, the goal is to encode the problem as a logic program so that answer sets of the program correspond to sequences of leaves with the cost less than or equal to a given integer. Below we will present several encodings. Some concepts are common among all of them.

We introduce these common concepts now. 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

$$\begin{array}{l} num(n) \\ maxCost(m). \end{array}$$

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

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

where i is an integer from 1 to n that 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

$$\begin{array}{l} \mathit{position}(X) \leftarrow X = 0..N-1, num(N) \\ \mathit{coloredPos}(X) \leftarrow X = 1..N-1, num(N), \end{array}$$

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

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:

$$\begin{array}{l} 1\{\mathit{leafPos}(L, P) : \mathit{position}(P)\}1 \leftarrow \mathit{leaf}(L) \\ \leftarrow \mathit{leafPos}(L, P),\ \mathit{leafPos}(L', P),\ \mathit{position}(P),\ L \neq L'. \end{array}$$

Intuitively, the first rule says that each leaf is assigned exactly one position. The second rule ensures that no two leaves are given the same position.

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

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

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

$$\begin{array}{rl} \mathit{posCost}(0, \mathit{Cost}) \leftarrow & \mathit{leafWeightCard}(L, \mathit{Cost}, C), \mathit{leafPos}(L, 0) \\ \mathit{posCost}(P, \mathit{Cost}) \leftarrow & \mathit{coloredPos}(P),\ \mathit{posColor}(P, \mathit{red}),\ \mathit{leafPos}(L, P), \\ & \mathit{leafWeightCard}(L, W, C),\ \mathit{posCost}(P-1, \mathit{Cost}'), \\ & \mathit{Cost} = \mathit{Cost}' + W \end{array} \tag{2}$$

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:

$$\begin{array}{rl} \mathit{seqCost}(1, \mathit{Cost}) \leftarrow & \mathit{posCost}(1, \mathit{Cost}) \\ \mathit{seqCost}(P, \mathit{Cost}) \leftarrow & \mathit{coloredPos}(P),\ P > 1,\ \mathit{seqCost}(P-1, C), \\ & \mathit{posCost}(P, C'),\ \mathit{Cost} = C + C'. \end{array} \tag{3}$$

Consequently, an answer set contains the ground atom $seqCost(n-1,c)$ 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*:

$$exists \leftarrow seqCost(N-1, Cost),\ num(N),\ Cost \leq M, maxCost(M) \tag{4}$$

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:

$$\leftarrow\ not\ exists \tag{5}$$

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 (2) and (3) 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 follows

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

In other words, $val(l)$ denotes the smallest number associated with l (either its weight or cardinality).

The *sequence* encoding depends on the following observation.

Observation 1: Given an arbitrary sequence of leaves there is a simple condition that determines the color to assign to each leaf in a colored position so that any other color assignment results in a colored sequence with the same or higher cost.

Let l_1 and l_2 be two leaves in a sequence so that l_1 immediately precedes l_2. We define the *color number* of a leaf l_2 to be

$$colorNum(l_2) = min(l_2^w + l_2^c, cost(l_1) + val(l_2)).$$

We now note that a color assigned to leaf l_2 in accordance with the formula

$$color(l_2) = \begin{cases} green & \text{if } colorNum(l_2) = l_2^w + l_2^c \\ red & \text{otherwise, if } colorNum(l_2) = cost(l_1) + l_2^w \\ blue & \text{otherwise, if } colorNum(l_2) = cost(l_1) + l_2^c \end{cases}$$

guarantees that any other color assignment results in a colored sequence with the same or higher cost.

Observation 1 paves a way to the *sequence* encoding, SEQ, that builds upon the DECL encoding by replacing choice rule (1) with a set of "deterministic rules." For instance,

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

is one of the rules in this set. We note that $\mathit{leafValue}(l, v)$ is defined in a way that $v = \mathit{val}(l)$ where l is a leaf.

The other difference with the DECL encoding is that rules defining *posCost* predicate are simpler. 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.

Intuitively, the advantage of this encoding in comparison with the encoding DECL is a reduced search space as color assignment requires no choices.

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 $\mathit{val}(l') < \mathit{val}(l)$ then the sequence M' constructed from M by changing the order of l and l' has a smaller cost than M, i.e., $\mathit{cost}(M') < \mathit{cost}(M)$. We say that M' is a *mirror* sequence of M.

Observation 2 allows us to add the constraint

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

to the SEQ encoding. We denote the resulting program by SEQ+. We note that this constraint may eliminate a sequence M that is a solution to a given weighted-sequence problem. Yet, such sequence M is guaranteed to have a mirror sequence to which constraint (6) is not applicable.

The motivation behind extending the SEQ encoding with (6) is that it reduces the search space by placing additional constraints on the answer sets. We expect that the closer a maximum cost of a given weighted-sequence problem to an optimal one, the more important the presence of constraint (6) is.

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 0*th 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.

Partition Encoding: For a leaf l and a sequence N of non-green leaves, we say that a sequence $l\,N$ is sorted if the leaves in N are in the ascending order with respect to their values.

In all encodings discussed previously, the GENERATE part enumerates sequences of given leaves. Observation 3 suggests that one may formalize GENERATE differently. We now describe a *partition* encoding, PART, that is based on this idea. Let n be the number of leaves in a given weighted-sequence problem. Instead of generating a sequence of n leaves, we can generate partitions that can be used to form such a sequence. Once the partitions are generated, they can be viewed as determining the sequence, in which they occur sorted according to the costs of their green nodes, with the 0th partition placed at the front (cf. Observation 2). We now specify the corresponding generation process:

– assign (arbitrarily) some leaves (up to $n - 1$) a green color (each of these leaves "marks" its partition),
– decide (arbitrarily) for each non-green leaf to which partition it belongs: either 0th or a green partition marked by some green node,
– assign (deterministically) each non-green leaf a position in its partition so that the resulting partition is sorted,
– ensure that the 0th partition is nonempty.

We now start the description of the TEST part. First, we insist (similarly to the SEQ++ encoding) that only a sequence M built from partitions so that the green nodes in M are sorted may form an answer set.

Next, we use the rules in (3), (4), and (5) to ensure that the resulting sequence is indeed a solution. To do so we introduce a number of rules into the DEFINE part of a program that together map a position of a leaf in its partition to a position of this leaf in the sequence (resulting from the generated partitions).

Finally, we use Observation 1 to support additional restrictions that forbid sequences where changing a color of some node from green (non-green) to non-green (green) produces sequences with smaller costs.

To implement these ideas in PART we used aggregates *#count* and *#sum* provided by the CLINGO language. The construct *#count* was used in three rules and the construct *#sum* in one rule of the encoding.

We note that the encoding PART is similar in its nature to the encoding SEQ++ as both programs incorporate similar restrictions (based on Observation 1-3) on the solution space. The main difference is in GENERATE where PART is more restrictive in choices. Also, the PART encoding is more sophisticated. For instance, it encodes sorting of a partition by simulating a procedural sorting algorithm based on computing ranks and exploiting the aggregate *#count* provided by CLINGO rather than enforcing sortedness by the appropriate constraint.

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 (CLINGO) 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 and SEQ*[6] in the CLINGCON language.

We note that only the rules containing functional predicates *posCost* and *seqCost* were changed in DECL and SEQ* to produce CLINGCON programs. For instance, the rules in (3) and (4) 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 (2), 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.

[6] By SEQ* we denote a set of *sequence* encodings: SEQ, SEQ+, and SEQ++.

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.

5 Experimental Analysis

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

Experiments were performed concurrently on three identical machines, each possessing 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. The process is as follows.

Given n and v, 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 we used $v = 12$ and $n \in \{8, 10, 11, 12\}$. We refer to these instances as *small* because of the low value of v.

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*, that we have considered. Clearly, all of them are satisfiable.

The process for creating harder instances involved an encoding and a solver. The encoding DECL was chosen along with CLINGO. We proceeded by starting with an instance $[S, m]$ in the set of easy instances. We used CLINGO 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 we had two pieces of information: $\hat{m}$ was optimal with respect to S, and $\hat{m}-1$ made the set S "barely" unsatisfiable. Instances obtained in this way from the *easy* instances formed the sets of *optimal* and *unsatisfiable* instances, respectively, while the instances $[S, \hat{m}+5]$ formed the set of *hard* instances.

To benchmark each encoding we used them to solve each of the four instance sets (easy, hard, optimal, and unsatisfiable). A time limit of 1500 seconds (25:00 minutes) was enforced for each instance. From each solve the grounding time, solving time, solution, and number of choices made were recorded for further study.

We now present and discuss the results of our experiments. Due to space limits only some results are presented here. For more results, we refer to `http://www.csr.uky.edu/WeightedSequence/`.

First, we note that CLINGCON was designed to address the "grounding bottleneck" of ASP. It attempts to accomplish that by performing a partial grounding only, that is, not instantiating all variables but leaving the task of finding appropriate values for them to a constraint solver. On the one hand, our results seem to indicate that CLINGCON succeeded. Indeed, the grounding times we collected for the four groups of instances we considered clearly show the superiority of CLINGCON over CLINGO in this respect (cf. `http://www.csr.uky.edu/WeightedSequence/`).

However, in ASP grounding is only a part of the story and as one comes to solving, the picture arguably is different. We will now focus on solving. In the tables we discuss below we report total times that include both the time for grounding and solving. We do so because ultimately it is the total time that is the right measure of the performance of a program solving tool in ASP.

From Easy to Hard or Clingo vs Clingcon: Tables 1 - 4 present the timing results on instances that are easy, hard, optimal, and unsatisfiable, respectively. First, the results suggest the soundness of our approach to generate increasingly harder instances by lowering the bound for the total weight. Indeed, for both solvers the tables show growing running times as we move across the four groups of instances from easy to unsatisfiable ones. More interestingly, the results suggest observations important for a comparison of the strengths and weaknesses of CLINGO and CLINGCON.

When run on easy instances, CLINGO and CLINGCON show similar performance (Table 1), with CLINGCON having a slight advantage (a significant one on the *partition* encodings). The reason is that grounding takes more time for CLINGO but problems are easy enough that CLINGCON can still solve them very quickly.

In the following tables "to" stands for "timeout".

Table 1. Easy Small Instances

	CLINGO					CLINGCON				
Instance	DECL	SEQ	SEQ+	SEQ++	PART	DECL	SEQ	SEQ+	SEQ++	PART
1-8-127	0.28	0.21	0.23	0.61	0.50	0.01	0.03	0.02	0.03	0.08
2-8-63	0.15	0.10	0.11	0.11	0.53	0.01	0.03	0.02	0.06	0.13
3-8-186	0.68	0.48	0.49	0.82	0.83	0.00	0.03	0.02	0.04	0.06
4-10-248	0.94	0.96	0.99	4.39	22.90	0.02	0.06	0.06	0.46	3.14
5-10-194	0.64	0.64	0.68	5.91	50.86	0.02	0.04	0.04	0.07	1.62
6-10-155	0.61	0.49	0.53	0.78	7.75	0.02	0.03	0.03	0.05	0.18
7-11-139	0.59	0.53	0.60	1.21	66.51	0.02	0.06	0.27	0.84	6.57
8-11-168	0.90	0.83	0.93	2.73	30.30	0.02	0.08	0.04	1.38	0.52
9-11-152	0.54	0.52	0.58	2.67	1.54	0.02	0.06	0.13	3.08	1.65
10-12-140	0.83	0.55	0.66	1.13	36.91	0.02	0.10	0.06	0.99	2.05

The situation changes when we consider hard instances (Table 2). Now clingcon, while still quite often outperforming clingo, in many cases does worse or much worse. In fact, clingcon times out 16% of the time.

Table 2. Hard Small Instances

	clingo					clingcon				
Instance	DECL	SEQ	SEQ+	SEQ++	PART	DECL	SEQ	SEQ+	SEQ++	PART
1-8-71	0.27	0.16	0.17	0.15	0.44	0.32	0.02	0.04	0.03	1.34
2-8-27	0.06	0.06	0.07	0.09	0.54	1.09	0.16	0.25	0.32	1.06
3-8-81	0.38	0.67	0.68	0.47	2.54	385.16	4.29	1.24	0.40	0.21
4-10-78	1.04	3.40	2.59	1.93	17.12	to	15.02	30.52	28.74	18.50
5-10-79	0.24	0.38	0.54	1.58	127.81	56.85	0.18	0.03	0.11	5.26
6-10-70	0.29	0.36	0.60	1.88	30.48	to	7.90	28.62	0.18	25.74
7-11-73	2.55	0.45	0.52	2.59	94.97	to	519.86	171.16	87.06	49.52
8-11-44	1.17	1.24	1.21	3.82	379.62	to	0.22	86.04	to	114.88
9-11-86	0.54	0.56	0.36	4.00	27.37	to	0.60	0.30	33.76	4.42
10-12-51	0.48	0.45	0.78	3.48	145.84	to	243.22	to	89.48	103.02

This trend continues when the two solvers are run on instances with the weight bound set at the optimal value (Table 3), and on instances that are “barely” unsatisfiable, the weight bound being set at 1 less the optimal value. In the first case clingcon times out in 46% of the 50 cases, while clingo terminates on each instance and under each encoding. In the second case, clingcon times out in 66% of cases while clingo times out only 4% of the time.

Table 3. Optimal Small Instances

	clingo					clingcon				
Instance	DECL	SEQ	SEQ+	SEQ++	PART	DECL	SEQ	SEQ+	SEQ++	PART
1-8-66	0.68	0.68	0.29	0.19	1.14	495.37	0.60	0.06	1.76	1.20
2-8-22	0.10	0.06	0.06	0.11	0.28	553.80	2.83	3.70	2.70	16.44
3-8-76	1.41	1.06	1.52	1.31	11.48	to	2.93	22.70	15.47	2.81
4-10-73	50.03	43.54	20.19	17.50	48.46	to	675.65	1079.84	to	3.68
5-10-74	1.23	5.64	5.19	11.11	327.94	to	53.65	23.27	763.50	403.82
6-10-65	23.26	16.59	8.14	3.26	144.99	to	304.59	1225.43	to	59.92
7-11-68	7.57	102.06	18.07	21.90	163.95	to	to	to	to	to
8-11-39	0.67	5.43	29.15	13.48	836.16	to	1412.74	to	to	to
9-11-81	19.82	23.33	25.16	32.60	713.39	to	522.50	to	to	to
10-12-46	1.87	6.41	22.18	18.04	1413.56	to	766.77	to	to	to

The results suggest that as the problems get harder (constraints get tighter and harder to satisfy), the initial advantage of clingcon that results from simplified grounding disappears. For hard problems, clingcon’s constraint solver or the way clingcon integrates the constraint solver with its ASP solver are not optimized enough to handle them. In contrast, the highly optimized ASP solver of clingo is still powerful enough to handle them.

Table 4. Unsatisfiable Small Instances

	clingo					clingcon				
Instance	decl	seq	seq+	seq++	part	decl	seq	seq+	seq++	part
1-8-65	5.99	2.95	3.51	1.84	19.96	to	43.77	39.17	12.39	83.65
2-8-21	0.12	0.12	0.12	0.13	3.57	1422.32	16.76	16.66	12.34	24.64
3-8-75	3.08	3.01	2.31	2.56	19.61	to	46.51	24.34	22.66	73.61
4-10-72	50.68	53.63	41.90	18.62	273.76	to	to	to	to	to
5-10-73	64.26	105.47	78.81	16.01	399.21	to	to	to	1085.86	1294.15
6-10-64	23.40	14.50	8.36	2.98	122.62	to	to	to	957.29	909.54
7-11-67	64.07	125.61	87.60	19.75	1311.52	to	to	to	to	to
8-11-38	100.57	129.44	73.57	19.67	1306.37	to	to	to	to	to
9-11-80	1353.04	71.56	87.32	27.70	to	to	to	to	to	to
10-12-45	60.42	88.24	39.00	22.83	to	to	to	to	to	to

From Declarative to Sophisticated: Next, we considered the effect of the encoding on the performance of solvers. The declarative encoding represents a rather direct and literal modeling of problem requirements in terms of rules. In particular, it does not take advantage of the observations that form the basis for the sequence and partition encodings, and that aim at constructing programs defining smaller search spaces of feasible candidates. Interestingly, despite that, for clingo the declarative, the encodings seq and seq+ are quite comparable, with no encoding being significantly better than the other two. In fact, we have analyzed the number of choices made by clingo on unsatisfiable instances and they correlate well with running times of the system (Table 5 illustrates the claim). This seems to suggest that clingo, and to be more precise, its mechanism to learn additional clauses restricting the search space, can compensate for at least some constraints that are not given explicitly.

As concerns clingcon, providing constraints explicitly is essential; clingcon times out on all but one declarative unsatisfiable instances. However, it terminates on three out of 10 instances under seq and seq+ encodings, and on six out of 10 instances under the seq++ encoding and five out of 10 instances under the part encoding, all four encodings providing additional "human" inferred constraints. These results suggest that learning in clingcon is still rather ineffective. Unsatisfiable instances are particularly relevant here as they force solvers to "consider" the entire search space.

Next, we note that understanding the problem and its constraints is not enough. Much still depends on how the necessary concepts are modeled. The encodings seq++ and part are designed so that the search spaces of candidate solutions they generate are isomorphic. However, the two encodings go about defining these search spaces quite differently. It turns out that the partition encoding is much less efficient (Table 5; we consider only clingo on unsatisfiable instances as clingcon timed out on most instances in that group). The partition encoding, part, uses the aggregate *sum* to model one of its constraints. We have also considered an alternative partition encoding, part-no-sum, that eliminates a rule with the aggregate *sum* in part using normal rules. Our results show that using this aggregate does not significantly change the number of choice points

Table 5. CLINGO: Unsat Instances; Reporting running time (choice points)

Instance	SEQ++	PART	PART-NO-SUM
1-8-65	1.84(7509)	19.96(28106)	10.86(31983)
2-8-21	0.13(1017)	3.57(6576)	0.69(4874)
3-8-75	2.56(7025)	19.61(21844)	9.07(17886)
4-10-83	18.62(27769)	273.76(118054)	106.77(150284)
5-10-61	16.01(25478)	399.21(205398)	112.40(201958)
6-10-64	2.98(8685)	122.62(55158)	33.24(62826)
7-11-67	19.75(35376)	1311.52(404169)	285.40(402663)
8-11-38	19.67(35852)	1306.37(402672)	231.77(420197)
9-11-80	27.70(45504)	to	936.19(1440642)
10-10-72	22.83(41926)	to	396.34(591053)

reported by CLINGO on PART and PART-NO-SUM. However, the PART encoding results in overall higher processing times. We also note that the number of choice points on SEQ++ is substantially smaller than for PART and PART-NO-SUM. Both PART and PART-NO-SUM contain rules that use the aggregate *count*. Incidentally, the encoding SEQ++, which seems to perform best of the three does not use aggregates at all. While these results are based on too small a sample to derive any general conclusions, they do suggest that arbitrary use of aggregates may be detrimental.

6 Conclusions and Future Work

Our experimental findings lead us to a number of observations. Highly tuned ASP search procedures (in particular, CLINGO) display a similar behavior on both "literal" (DECL) and "sophisticated" (SEQ++) encodings of a weighted-sequence problem. The sophisticated encoding imposes a number of restrictions on the problem's search space in comparison with the literal encoding. For a hybrid system such as CLINGCON (that combines both ASP and CSP techniques in its search), reduced search space is crucial for the system's scalability. Overall our key finding is that the effectiveness of the search procedure used by CLINGCON lags behind that of CLINGO. CASP promises to become a milestone in declarative problem solving by providing the means of solving ASP grounding bottleneck. To do so, solid performance of CSP solvers and good communication between ASP and CSP sides are essential.

Currently we are constructing instances of the weighted-sequence problem with large weights. We will use these instances to investigate the effectiveness of CLINGCON in processing them. We will focus on the question how close to optimal solutions will CLINGCON be able to get. We also plan to evaluate the performance of CSP/CLP systems on the weighted-sequence problem. This will provide us with a better understanding on how ASP, CASP, and pure CSP/CLP systems compare.

Ultimately we will shift our attention to the query optimization problem of finding an optimal join order using the state-of-the-art ASP/CASP/CSP technology. We expect that the experience and knowledge gained during our work on weighted-assignment problem will be instrumental.

Acknowledgments

We are grateful to Philip Cannata for bringing the problem of finding an optimal join order in query optimization to our attention, and to Vladimir Lifschitz and Yuanlin Zhang for useful discussions related to the topic of this work. 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. Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., Pfeifer, G.: Aggregate functions in disjunctive logic programming: semantics, complexity, and implementation in DLV. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003). pp. 847–852. Morgan Kaufmann (2003)
3. Gebser, M., Schaub, T., Thiele, S.: Gringo: A new grounder for answer set programming. In: Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning. pp. 266–271 (2007)
4. 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_gui%de/2010-10-04/guide.pdf`
5. 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)
6. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In: Proceedings of ICLP-2009. pp. 235–249 (2009)
7. 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)
8. Lewis, J.: Cost-Based Oracle Fundamentals. Apress (2005)
9. Lifschitz, V.: Answer set programming and plan generation. Artif. Intell. 138(1-2), 39–54 (2002)
10. 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)
11. Mellarkod, V.S., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. Annals of Mathematics and Artificial Intelligence (2008)
12. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence 25, 241–273 (1999)
13. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). Journal of the ACM 53(6), 937–977 (2006)
14. Steinbrunn, M., Moerkotte, G., Kemper, A.: Heuristic and randomized optimization for the join ordering problem. The VLDB Journal (1997)