# A Comparison of CLP(FD) and ASP Solutions to NP-Complete Problems

Agostino Dovier[1], Andrea Formisano[2], and Enrico Pontelli[3]

[1] Univ. di Udine, Dip. di Matematica e Informatica
dovier@dimi.uniud.it
[2] Univ. di L'Aquila, Dip. di Informatica
formisano@di.univaq.it
[3] New Mexico State University, Dept. Computer Science
epontell@cs.nmsu.edu

**Abstract.** This paper presents experimental comparisons between declarative encodings of various computationally hard problems in both Answer Set Programming (ASP) and Constraint Logic Programming (CLP) over finite domains. The objective is to identify how the solvers in the two domains respond to different problems, highlighting strengths and weaknesses of their implementations and suggesting criteria for choosing one approach versus the other. Ultimately, the work in this paper is expected to lay the ground for transfer of concepts between the two domains (e.g., suggesting ways to use CLP in the execution of ASP).

## 1 Introduction

The objective of this work is to experimentally compare the use of two distinct logic-based paradigms in solving computationally hard problems. The two paradigms considered are *Answer Set Programming (ASP)* [2] and *Constraint Logic Programming over Finite Domains (CLP(FD))* [19]. The motivation for this investigation arises from the successful use of both paradigms in dealing with various classes of combinatorial problems, and the need to better understand their respective strengths and weaknesses. Ultimately, we hope this work will provide indication for integration and cooperation between the two paradigms (e.g., along the lines of [8]).

It is well-known [17,2] that, given a propositional normal logic program $P$, deciding whether or not it admits an *answer set* [11] is an NP-complete problem. As a consequence, any NP-complete problem can be encoded as a propositional normal logic program under answer set semantics. Answer-set solvers [22] are programs designed for computing the answer sets of normal logic programs; these tools can be seen as theorem provers, or model builders, enhanced with several built-in heuristics to guide the exploration of the solution space. Most ASP solvers rely on variations of the Davis-Putnam-Longeman-Loveland procedure in their computations. Such solvers are often equipped with a front-end that transforms a collection of non-propositional normal clauses (with limited use of function symbols) in a *finite* set of ground instances of such clauses. Some solvers provide also classes of *optimization statements*, used to select answer sets that maximize or minimize an objective function dependent on the content of the answer set.

An alternative framework, frequently adopted to handle NP-complete problems, is *Constraint Logic Programming over Finite Domains* [13,19]. In this context, a finite

M. Gabbrielli and G. Gupta (Eds.): ICLP 2005, LNCS 3668, pp. 67–82, 2005.

domain of objects (typically integers) is associated to each variable in the problem specification, and the typical constraints are literals of the forms $s = t$, $s \neq t$, $s < t$, $s \leq t$, where $s$ and $t$ are arithmetic expressions. Encodings of NP-complete problems and of search strategies are very natural and declarative in this framework. Indeed, a large literature has been developed presenting applications of CLP(FD) to a variety of search and optimization problems [19].

In this paper, we report the outcomes of preliminary experiments aimed at comparing these two declarative approaches in solving combinatorial problems. We address a set of computationally hard problems—in particular, we mostly consider decision problems known to be NP-complete. We formalize each problem, both in CLP(FD) and in ASP, by taking advantage of the specific features available in each logical frameworks, attempting to encode the various problems in the *most declarative* possible way. In particular, we adopt a *constraint-and-generate* strategy for the CLP code, while in ASP we exploit the usual *generate-and-test* approach. Wherever possible, we make use of solutions to these problems that have been presented and accepted in the literature.

With this work we intend to develop a bridge between these two logic-based frameworks, in order to emphasize the strengths of each approach and to promote cross-fertilizations. This study also complements the system benchmarking studies, that have recently appeared for both CLP(FD) systems [10,20] and ASP solvers [1,16,14].

## 2 The Experimental Framework

In this paper we report on the experimentation we conducted by using one CLP(FD) implementation and two ASP-solvers. The CLP programs have been designed for execution by SICStus Prolog 3.11.2 (using the library `clpfd`)—though the code is general enough to be used on different platforms (e.g., BProlog, ECLiPSe, GNU-Prolog) with minor syntactic adjustment [23].The ASP programs have been designed to be processed by *lparse*, the grounding preprocessor adopted by both the SModels (version 2.28) and the CModels (version 3.03) systems [22]. The CModels system makes use of a SAT solver to compute answer sets—in our experiments we selected the default underlying SAT solver, namely mChaff.

We focused on well-known computationally-hard problems. Among them: Graph $k$-coloring (Section 3), Hamiltonian circuit (Section 4), Schur numbers (Section 5), protein structure prediction on a 2D lattice [3] (Section 6), planning in a block world (Section 7), and generalized Knapsack (Section 8). Observe that, while some of the programs have been drawn from the best proposals appeared in the literature, others are novel solutions, developed in this project (e.g., the ASP implementation of the PF problem and the planning implementation in CLP(FD)).

In the remaining sections of this paper, we describe the solutions to the various problems and report the results from the experiments. All the timing results, expressed in seconds, have been obtained by measuring only the CPU usage time needed for computing the first solution, if any—thus, we ignore the time spent in reading the input, as well as the time spent to ground the program, in the case of the ASP solvers. We used the `runtime` option to measure the time in CLP(FD), that does not account for the time spent for garbage collection and for system calls. All tests have been performed on a

PC (P4 processor 2.8 GHz, and 512 MB RAM memory) running Linux kernel 2.6.3. Complete codes and results (as well as encodings of other problems) are reported in [7].

## 3 $k$-Coloring

The $k$-coloring problem computes the coloring of a graph using $k$ colors. The main source of case studies adopted in our experiments is the repository of "Graph Coloring and its Generalizations" [24], which provides a rich collection of instances, mainly aimed at benchmarking algorithms and approaches to graph problems. Let us describe the two formalizations of $k$-coloring.

**CLP(FD):** In this formulation, we assume that the input graph is represented by a single fact of the form `graph([1,2,3],[[1,2],[1,3],[2,3]])`, where the first argument represents the list of nodes, while the second argument is the list of edges. This is a possible constrain-and-generate CLP(FD)-encoding of $k$-coloring:

```
coloring(K, Output) :- graph(Nodes, Edges),
      create_output(Nodes, Colors, Output), domain(Colors,1, K),
      different(Edges, Output), labeling([ff], Colors).
create_output([],[],[]).
create_output([N|Nodes], [C|Colors], [N-C|Output]) :-
      create_output(Nodes, Colors, Output).
different([],_).
different([[A,B]|R], Output) :- member(A-CA, Output),
      member(B-CB, Output), CA #\= CB, different(R, Output).
```

In this program, `Output` is intended to be a list of pairs of variables `N-C` where, for each node `N` we introduce a color variable `C` in the range `1...K`. The predicate `different` imposes disequality constraints between variables related to adjacent nodes. We used the `ff` option of `labeling` since it offered the best results for this problem.

**ASP:** Regarding the ASP encoding of $k$-coloring we adopt a different representation for graphs. Each node `V` is described by a fact `node(V).` If nodes are natural numbers, a compact interval notation is allowed (e.g., `node(1..138)`). Edges are described by facts, e.g., `edge(1,36). edge(2,45). edge(138,36).`

A natural ASP encoding of the $k$-coloring problem is:

```
(1)     col(1..k).
(2)     :- edge(X,Y), col(C), color(X,C), color(Y,C).
(3)     1 {color(X,C):col(C)} 1 :- node(X).
```

Rule `(1)` states that there are `k` colors (`k` is a constant to be initialized during the grounding stage). The ASP-constraint `(2)` asserts that two adjacent nodes cannot have the same color, while `(3)` states that each node has exactly one color. Note that, by using domain restricted variables, the single ASP-constraint `(2)` states the property that two adjacent nodes cannot have the same color for all edges $\langle X, Y \rangle$. The same property is described by the predicate `different` in the CLP(FD) code, but in that case a recursive definition is required. This fact shows a common situation that will be observed again in the following sections: ASP often permits a significantly more compact encoding of the problem w.r.t. CLP(FD).

**Table 1.** Graph $k$-coloring ('–' denotes no answer in at least 30 minutes of CPU-time— '?' means that none of the three solvers gave an answer)

| Instance | | 3-colorability | | | | 4-colorability | | | | 5-colorability | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Graph | V×E | | SModels | CModels | CLP(FD) | | SModels | CModels | CLP(FD) | | SModels | CModels | CLP(FD) |
| 1-FullIns_5 | 282×3247 | N | 1.06 | 0.15 | 0.10 | N | – | 0.23 | 2.90 | N | – | 107.78 | – |
| 4-FullIns_4 | 690×6650 | N | 0.94 | 0.29 | 0.46 | N | 2.20 | 0.35 | 1.98 | N | 10.02 | 0.42 | – |
| 5-FullIns_4 | 1085×11395 | N | 1.72 | 0.47 | 1.26 | N | 4.67 | 0.57 | 3.58 | N | 23.79 | 0.70 | – |
| 3-FullIns_5 | 2030×33751 | N | 5.92 | 1.23 | 7.24 | N | 21.31 | 1.51 | 13.69 | N | – | 1.96 | – |
| 4-FullIns_5 | 4146×77305 | N | 15.11 | 2.69 | 33.44 | N | 69.30 | 3.37 | 42.53 | N | 414.93 | 4.19 | – |
| 3-Insertions_3 | 56×110 | N | 4.28 | 4.16 | 1281.18 | Y | 0.03 | 0.04 | <0.01 | Y | 0.04 | 0.04 | <0.01 |
| 4-Insertions_3 | 79×156 | N | 328.25 | 1772.14 | – | Y | 0.05 | 0.04 | <0.01 | Y | 0.06 | 0.05 | <0.01 |
| 2-Insertions_4 | 149×541 | N | 1.20 | 0.15 | 2.04 | ? | – | – | – | Y | 0.25 | 0.07 | 0.01 |
| 4-Insertions_4 | 475×1795 | N | – | 1443.33 | – | ? | – | – | – | Y | 3.402 | 0.32 | – |
| DSJR500.1 | 500×3555 | N | 0.53 | 0.18 | 0.18 | N | 2.78 | 0.21 | 0.18 | N | – | 0.26 | 0.19 |
| DSJC500.1 | 500×12458 | N | 2.19 | 0.45 | 0.64 | N | 12.30 | 0.57 | 0.76 | N | 6328.45 | 6.21 | 46.55 |
| DSJR500.5 | 500×58862 | N | 25.76 | 1.81 | 2.97 | N | 175.63 | 2.26 | 2.98 | N | 971.46 | 2.71 | 3.09 |
| DSJC500.5 | 500×62624 | N | 28.29 | 1.92 | 3.15 | N | 376.35 | 2.36 | 3.19 | N | 2707.64 | 2.84 | 3.47 |
| flat300_20_0 | 300×21375 | N | 6.39 | 0.68 | 0.63 | N | 86.91 | 0.84 | 0.64 | N | 1555.37 | 1.08 | 0.69 |
| flat300_26_0 | 300×21633 | N | 6.45 | 0.70 | 0.65 | N | 131.91 | 0.87 | 0.67 | N | 3711.80 | 1.13 | 0.69 |
| flat300_28_0 | 300×21695 | N | 6.51 | 0.70 | 0.65 | N | 34.76 | 0.86 | 0.69 | N | 322.99 | 1.02 | 0.67 |
| fpsol2.i.1 | 496×11654 | N | 2.75 | 0.41 | 0.77 | N | 24.98 | 0.52 | 0.77 | N | 205.12 | 0.61 | 0.84 |
| fpsol2.i.2 | 451×8691 | N | 1.92 | 0.33 | 0.53 | N | 16.66 | 0.40 | 0.54 | N | 279.96 | 0.52 | 0.55 |
| fpsol2.i.3 | 425×8688 | N | 1.91 | 0.32 | 0.5 | N | 16.63 | 0.40 | 0.51 | N | 277.91 | 0.49 | 0.51 |
| gen200_p0.9_44 | 200×17910 | N | 5.53 | 0.57 | 0.36 | N | 30.87 | 0.70 | 0.36 | N | 306.81 | 0.84 | 0.38 |
| gen200_p0.9_55 | 200×17910 | N | 5.54 | 0.57 | 0.36 | N | 39.56 | 0.71 | 0.36 | N | 287.14 | 0.85 | 0.38 |
| gen400_p0.9_55 | 400×71820 | N | 38.91 | 2.19 | 2.88 | N | 656.07 | 2.68 | 2.89 | N | 4892.74 | 3.24 | 2.93 |
| gen400_p0.9_65 | 400×71820 | N | 39.02 | 2.16 | 2.88 | N | 275.33 | 2.67 | 2.87 | N | 1563.52 | 3.22 | 2.92 |
| wap05a | 905×43081 | N | 11.39 | 1.38 | 2.96 | N | 62.81 | 1.73 | 2.96 | N | 949.66 | 2.07 | 2.96 |
| wap06a | 947×43571 | N | 11.63 | 1.42 | 3.25 | N | 62.70 | 1.75 | 3.24 | N | 1326.84 | 2.13 | 3.26 |
| wap07a | 1809×103368 | N | 31.98 | 3.28 | 15.14 | N | 191.06 | 4.12 | 15.14 | N | 2861.64 | 4.99 | 15.19 |
| wap08a | 1870×104176 | N | 32.07 | 3.31 | 16.17 | N | 192.54 | 4.15 | 16.22 | N | 3604.96 | 5.08 | 16.18 |

**Table 2.** The $M$-$N$-Queens problem ('–' denotes no answer in 10 min. of CPU-time)

| Instance | | Solvability for $M = N - 1$ | | | | | Solvability for $M = N$ | | | | | Solvability for $M = N + 1$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $N$ | $V \times E$ | | SModels | CModels | CLP(FD) | ugraphs | | SModels | CModels | CLP(FD) | ugraphs | | SModels | CModels | CLP(FD) | ugraphs |
| 5 | 25×320 | N | 0.06 | 0.07 | 0.01 | <0.01 | Y | 0.06 | 0.07 | <0.01 | <0.01 | Y | 0.07 | 0.08 | <0.01 | <0.01 |
| 6 | 36×580 | N | 1.00 | 0.11 | 0.01 | <0.01 | N | 63.80 | 198.65 | 1.33 | 0.02 | Y | 0.66 | 0.19 | <0.01 | 0.16 |
| 7 | 49×952 | N | 341.17 | 0.20 | 0.02 | 0.03 | Y | 1.95 | 0.18 | <0.01 | 0.29 | Y | 0.54 | 14.08 | 0.02 | 0.35 |
| 8 | 64×1456 | N | – | 0.42 | 0.16 | 0.89 | N | – | – | – | 224.11 | Y | 116.50 | 1.28 | 1.04 | 807.22 |
| 9 | 81×2112 | N | – | 0.85 | 1.37 | 106.64 | ? | – | – | – | – | Y | – | – | 138.85 | 131.27 |
| 10 | 100×2940 | N | – | 3.63 | 14.53 | – | ? | – | – | – | – | ? | – | – | – | – |
| 11 | 121×3960 | N | – | 10.62 | 148.74 | – | ? | – | – | – | – | ? | – | – | – | – |

**Results:** We tested the above programs on more than one hundred instances drawn from [24]. Such instances belong to various classes of graphs which come from different sources in the literature. Table 1 shows an excerpt of the results we obtained for $k$-coloring with $k = 3, 4, 5$. The columns report the time (in seconds) using the three systems; the first column of each result indicates whether a solution exists for the problem instance.

A particular class of graph coloring problems listed in [24] originates from encoding a generalized form of the $N$-queens problem. Graphs for the $M$-$N$-queen problems are obtained as follows. The nodes correspond to the cells of a $N \times N$ chess-board. Two nodes $u$ and $v$ are connected by an (undirected) edge if a queen in the cell $u$ attacks the

cell $v$. Solving the $M$-$N$-queens problem consists of determining whether such graph is $M$-colorable. In the particular case where $M = N$, this is equivalent to finding $N$ independent solutions to the classical $N$-queens problem. Observe that, for $M < N$ the graph cannot be colored. We ran a number of tests on this specific class of graphs. Table 2 lists the results obtained for $N = 5, \ldots, 11$ and $M = N - 1, N, N + 1$. For the sake of completeness, we also experimented, on these instances, using the library `ugraphs` of SICStus Prolog (a library independent from the library `clpfd`), where the `coloring/3` predicate is provided as a built-in feature. `ugraphs` is slower than CLP(FD) for small instances, however, it finds solutions in acceptable time for some larger instances, whereas CLP(FD) times out.

## 4 Hamiltonian Circuit

In this section we deal with the problem of establishing whether a directed graph admits an Hamiltonian circuit. The graph representations adopted are the same as in the previous section, with the restriction that graph nodes are `1..N` (needed to correctly use the built-in predicate `circuit` of SICStus Prolog).

**CLP(FD):** A possible CLP(FD) encoding is the following:

```
hc(Path) :- graph(Nodes, Edges), length(Nodes, N),
     length(Path, N), domain(Path, 1, N),
     make_domains(Path, 1, Edges, N),
     circuit(Path), labeling([ff], Path).
make_domains([], _, _, _).
make_domains([X|Y], Node, Edges, N) :-
     findall(Z, member([Node,Z], Edges), Successors),
     reduce_domains(N, Successors, X),
     Node1 is Node+1, make_domains(Y, Node1, Edges, N).
reduce_domains(0, _, _) :- !.
reduce_domains(N, Successors, Var) :- N>0, member(N,Successors),
     !, N1 is N-1, reduce_domains(N1, Successors, Var).
reduce_domains(N, Successors, Var) :-
     Var #\= N, N1 is N-1, reduce_domains(N1, Successors, Var).
```

We use the built-in predicate `circuit`, provided by `clpfd` in SICStus. In the literal `circuit(List)`, the `List` is a list of domain variables or integers. The goal `circuit`$([X_1, \ldots, X_n])$ constrains the variables so that the set of edges $\langle 1, X_1\rangle$, $\langle 2, X_2\rangle, \ldots, \langle n, X_n\rangle$ is an Hamiltonian circuit. The predicate `make_domains` restricts the admissible values for the variable $X_i$ to the successors of node $i$ in the graph.

**ASP:** The following program for Hamiltonian circuit comes from the ASP literature [18]:

```
(1)   1 {hc(X,Y) : edge(X,Y)} 1 :- node(X).
(2)   1 {hc(Z,X) : edge(Z,X)} 1 :- node(X).
(3)   reachable(X) :- node(X), hc(1,X).
(4)   reachable(Y) :- node(X), node(Y), reachable(X), hc(X,Y).
(5)   :- not reachable(X), node(X).
```

**Table 3.** Hamiltonian circuit ('–' denotes no answer within 30 minutes of CPU-time)

| Instance | node×edges | Hamiltonian? | | | |
|---|---|---|---|---|---|
| | | | SModels | CModels | CLP(FD) |
| hc1 | 200×1250 | Y | 2.99 | 37.59 | 0.34 |
| hc2 | 200×1250 | Y | 2.99 | 1394.15 | 0.34 |
| hc3 | 200×1250 | Y | 3.03 | 20.06 | 0.32 |
| hc4 | 200×1250 | Y | 2.98 | 93.10 | 0.34 |
| hc5 | 200×1250 | N | 1.44 | 0.22 | 0.24 |
| hc6 | 200×1250 | N | 1.44 | 0.21 | 0.10 |
| hc7 | 200×1250 | N | 1.44 | 0.20 | 0.25 |
| hc8 | 200×1250 | N | 1.44 | 0.20 | 0.26 |
| np10c | 10×90 | Y | 0.01 | 0.05 | 0.0 |
| np20c | 20×380 | Y | 0.07 | 0.82 | 0.0 |
| np30c | 30×870 | Y | 0.26 | 0.27 | 0.01 |
| np40c | 40×1560 | Y | 0.91 | 4.38 | 0.02 |
| np50c | 50×2450 | Y | 2.59 | 118.18 | 0.03 |
| np60c | 60×3540 | Y | 7.38 | 24.81 | 0.05 |
| np70c | 70×4830 | Y | 15.68 | 9.47 | 0.07 |
| np80c | 80×6320 | Y | 27.79 | 12.55 | 0.11 |
| np90c | 90×8010 | Y | 45.66 | 128.25 | 0.15 |

| Instance | node×edges | Hamiltonian? | | | |
|---|---|---|---|---|---|
| | | | SModels | CModels | CLP(FD) |
| 2xp30 | 60×316 | N | 0.14 | 0.02 | 0.03 |
| 2xp30.1 | 60×318 | Y | 0.18 | 4.61 | 0.02 |
| 2xp30.2 | 60×318 | Y | – | 2.69 | 5.38 |
| 2xp30.3 | 60×318 | Y | – | 2.70 | 5.38 |
| 2xp30.4 | 60×318 | N | – | – | – |
| 4xp20 | 80×392 | N | 0.24 | 0.04 | 0.04 |
| 4xp20.1 | 80×395 | N | – | 1.47 | 0.04 |
| 4xp20.2 | 80×396 | Y | 0.37 | 3.32 | 0.03 |
| 4xp20.3 | 80×396 | N | 0.24 | 2.65 | – |
| nv70a440 | 70×423 | Y | 0.28 | 1.33 | 0.05 |
| nv70a460 | 70×429 | Y | 0.28 | 3.00 | 0.03 |
| nv70a480 | 70×460 | Y | 0.29 | 1.66 | 0.06 |
| nv70a500 | 70×473 | Y | 0.29 | 1.73 | 0.03 |
| nv70a520 | 70×478 | Y | 0.29 | 0.36 | 0.05 |
| nv70a540 | 70×507 | Y | 0.31 | 4.19 | 0.04 |
| nv70a560 | 70×516 | Y | 0.32 | 0.62 | 0.05 |
| nv70a580 | 70×540 | Y | 0.32 | 1.00 | 0.04 |

The description of the search space is given by rules `(1)` and `(2)`: for each node `X`, exactly one outgoing edge `(X,Y)` and one incoming edge `(Z,X)` belong to the circuit (represented by the predicate `hc`). Rules `(3)` and `(4)` define the transitive closure of the relation `hc` starting from node number `1`. The "test" phase is expressed by the ASP-constraint `(5)`, which weeds out the answer sets that do not represent solutions to the problem. Also in this case, the ASP approach permits a more compact encoding (even if in CLP(FD) we exploited the built-ins `circuit` and `findall`).

**Results:** Most of the problem instances have been taken from the benchmarks used to compare ASP-solvers [16]. Graphs `hc1`–`hc8` are drawn from `www.cs.uky.edu/ai/benchmark-suite/hamiltonian-cycle.html`. All other graphs are chosen from `assat.cs.ust.hk/Assat-2.0/hc-2.0.html`. The graphs `np`$n$`c` are complete directed graphs with $n$ nodes and one edge $\langle u, v\rangle$ for each pair of distinct nodes. The graphs `nv`$v$`a`$a$ are randomly generated graphs, having at most $v$ nodes and $a$ edges. The instances `2xp30` (resp., `4xp20`) are obtained by joining 2 (resp., 4) copies of the graph `p30` (resp., `p20`) plus 2 (resp., 3–4) new edges. Graphs `p20` and `p30` are graphs provided in the SModels' distribution [22]. Table 3 lists the results.

## 5 Schur Numbers

A set $S \subseteq \mathbb{N}$ is *sum-free* if the intersection of $S$ and the set $S + S = \{x + y : x \in S, y \in S\}$ is empty. The *Schur number* $S(P)$ is the largest integer $n$ for which the set $\{1, \ldots, n\}$ can be partitioned in $P$ sum-free sets. For instance, $\{1, 2, 3, 4\}$ can be partitioned in $S_1 = \{1, 4\}$ and $S_2 = \{2, 3\}$. Observe that the sets $S_1 + S_1 = \{2, 5, 8\}$ and $S_2 + S_2 = \{4, 5, 6\}$ are sum-free. The set $\{1, 2, 3, 4, 5\}$, instead, originates at least 3 sum-free subsets, thus, $S(2) = 4$. It should be noted that, so far, only 4 Schur numbers have been computed, i.e., $S(1) = 1$, $S(2) = 4$, $S(3) = 13$, and $S(4) = 44$. The best known bound for $S(5)$ is $160 \leq S(5) \leq 315$ [21]. Here, we focus on the decision problem: is $S(P) \geq N$? Namely, we look for a function $B : \{1, \ldots, N\} \longrightarrow \{1, \ldots, P\}$ such that: $(\forall I \in \{1, \ldots, N\})(\forall J \in \{I, \ldots, N\})(B(I) = B(J) \rightarrow B(I + J) \neq B(I))$.

**Table 4.** Schur numbers ('–' denotes no answer within 30 minutes of CPU-time)

| Instance $\langle P, N\rangle$ | is $Schur(P) \geq N$? | SModels | CModels | CLP(FD) | Instance $\langle P, N\rangle$ | is $Schur(P) \geq N$? | SModels | CModels | CLP(FD) |
|---|---|---|---|---|---|---|---|---|---|
| $\langle 4, 43\rangle$ | Y | 0.27 | 0.25 | 0.03 | $\langle 5, 111\rangle$ | Y | – | 0.53 | 0.16 |
| $\langle 4, 44\rangle$ | Y | 0.29 | 3.37 | 0.56 | $\langle 5, 112\rangle$ | Y | – | 0.55 | 0.16 |
| $\langle 4, 45\rangle$ | N | 510.01 | 892.54 | 1204.86 | $\langle 5, 113\rangle$ | Y | – | 0.55 | 0.16 |
| $\langle 4, 46\rangle$ | N | 561.80 | 813.73 | 1340.64 | $\langle 5, 114\rangle$ | Y | – | 11.75 | 0.16 |
| $\langle 4, 47\rangle$ | N | 767.80 | 791.37 | 1473.02 | $\langle 5, 115\rangle$ | Y | – | 82.48 | 8.63 |
| $\langle 4, 48\rangle$ | N | 978.84 | 805.69 | 1565.28 | $\langle 5, 116\rangle$ | Y | – | 60.47 | 8.92 |
| $\langle 4, 49\rangle$ | N | 1258.57 | 679.20 | 1698.08 | $\langle 5, 117\rangle$ | Y | – | 762.91 | 9.74 |
| $\langle 5, 109\rangle$ | Y | – | 14.05 | 0.13 | $\langle 5, 118\rangle$ | Y | – | 21.84 | 10.19 |
| $\langle 5, 110\rangle$ | Y | – | 33.29 | 0.14 | $\langle 5, 119\rangle$ | Y | – | – | 66.95 |

**CLP(FD):** For doing that, in $CLP(FD)$ we introduce a list of constrained variables `List` $= [$`B1`$, \ldots,$ `BN`$]$ ranging on `1..P`.

```
schur(N,P) :- length(List,N), domain(List,1,P),List=[1,2|_],
              constraints(List,N), labeling([leftmost],List).
constraints(List, N) :- recursion(List,1,1,N).
recursion(_,I,_,N):- I>N, !.
recursion(List,I,J,N):- I+J>N,!,I1 is I+1,recursion(List,I1,1,N).
recursion(List,I,J,N):- I>J,!,J1 is J+1,recursion(List,I,J1,N).
recursion(List,I,J,N):- K is I+J, J1 is J+1,
      nth(I,List,BI), nth(J,List,BJ), nth(K,List,BK),
      (BI #= BJ) #=> (BK #\= BI), recursion(List,I,J1,N).
```

Each variable `Bi` in `List` can assume values in `1..P`. Its value identifies the block of the partition $i$ belongs to. The predicate `recursion` states that for all `I` and `J`, with `1` $\leq$ `I` $\leq$ `J` $\leq$ `N`, the numbers `I`, `J` and `I+J` must not be all in the same block. We set `B1 = 1` and `B2 = 2` to remove some simple symmetries, for a fair comparison w.r.t. the ASP solution that uses rules `(4)` and `(5)`—see below. We have chosen the `leftmost` option of labeling.

**ASP:** The function $B$ mentioned above is here implemented by a predicate `inpart(X,P)` representing the fact that number $X$ is assigned to part $P$:

```
(1) number(1..n).       part(1..p).
(2) 1 { inpart(X,P) : part(P) } 1 :- number(X).
(3) :- number(X;Y), part(P), X<=Y, inpart(X,P),
       inpart(Y,P), inpart(X+Y,P).
(4) :- number(X),part(P;P1),inpart(X,P),P1<P,not occupied(X,P1).
(5) occupied(X,P) :- number(X;Y), part(P), Y<X, inpart(Y,P).
```

Rule `(2)` states that `inpart` is a function from numbers to partitions. The ASP-constraints `(3)` states that, for any $X$ and $Y$, the three numbers $X$, $Y$, and $X + Y$ cannot belong to the same partition. The declarative formalization of the problem could be ended here. However, it is customary to add also the constraints `(4)` and `(5)`, that remove symmetries, by selecting the free partition with the lowest index.

**Results:** Table 4 reports the execution times we obtained. Let us observe that, unfortunately, we are still far from the best known lower bound of 160 for $S(5)$.

## 6 Protein Structure Prediction

Given a sequence $S = s_1 \cdots s_n$, with $s_i \in \{h, p\}$, the *2D HP-protein structure prediction problem* (reduced from [3]) is the problem of finding a mapping (*folding*) $\omega : \{1, \ldots, n\} \longrightarrow \mathbb{N}^2$ such that

$$(\forall i \in [1, n-1]\})\mathsf{next}(\omega(i), \omega(i+1)) \text{ and } (\forall i, j \in [1, n])(i \neq j \rightarrow \omega(i) \neq \omega(j))$$

and minimizing the energy:

$$\sum_{\substack{1 \leq i \leq n-2 \\ i+2 \leq j \leq n}} \mathsf{Pot}(s_i, s_j) \cdot \mathsf{next}(\omega(i), \omega(j))$$

where $\mathsf{Pot}(s_i, s_j) \in \{0, -1\}$ and $\mathsf{Pot} = -1$ if and only if $s_i = s_j = h$. The condition $\mathsf{next}(\langle X_1, Y_1\rangle, \langle X_2, Y_2\rangle)$ holds between two adjacent positions of a given lattice if and only if $|X_1 - X_2| + |Y_1 - Y_2| = 1$. Without loss of generality, we set $\omega(1) = \langle n, n\rangle$ and $\omega(2) = \langle n, n+1\rangle$, to remove some symmetries in the solution space. To remove all symmetries, we should require, that when the sequence turns for the first time, it turns, e.g., to the right. For the sake of simplicity we did not add this disjunctive constraint in the code. Instead, to reduce the solution's space in these experiments, we further add the heuristics constraints $X_i, Y_i \in [N - \sqrt{N}, N + \sqrt{N}]$.

Intuitively, we look for a self-avoiding walk that maximizes the number of contacts between occurrences of objects (aminoacids) of kind $h$ (see Figure 1). Contiguous occurrences of $h$ in the input sequence $S$ contribute in the same way to the energy associated to each spatial conformation and thus they are not considered in the objective function. Note that two objects can be in contact only if they are at an odd distance in the sequence (odd property of the lattice). This problem is a version of the protein structure prediction problem, whose decision problem is known to be NP-complete [4].

**CLP(FD):** A complete CLP(FD) encoding of this problem (based on the ideas in [3]) can be found in [7]. An extension of this code (in 3D, inside a realistic lattice, and with a more complex energy function) has been used to predict the spatial shape of real proteins [5]. In this case the labeling parameter chosen is `ff`.

**ASP:** As far as we know, there are no ASP formulations of this problem available in the literature. A specific instance of the problem is represented as a set of facts, describing the sequence of aminoacids. For instance, the protein denoted by `hpphpphpph` (or simply $(\texttt{hpp})^3\texttt{h}$ using regular expressions) is described as:

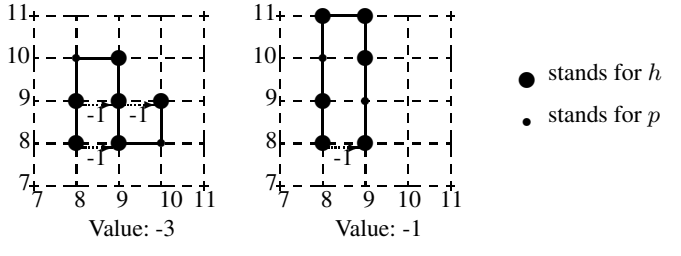


**Fig. 1.** Two foldings for $S = hhphhhph$ ($n = 8$). The leftmost one is minimal.

```
prot(1,h). prot(2,p). prot(3,p). prot(4,h). prot(5,p).
prot(6,p). prot(7,h). prot(8,p). prot(9,p). prot(10,h).
```

The ASP code is as follows:

```
(1) size(10).      %%% size(N) where N is input length
(2) range(7..13).  %%% [ N-sqrt{N}, N+sqrt{N} ]
(3) sol(1,N,N)   :- size(N).
(4) sol(2,N,N+1) :- size(N).
(5) 1 { sol(I,X,Y) : range(X;Y) } 1 :- prot(I,Amino).
(6) :- prot(I1,A1), prot(I2,A2), I1 < I2,
       sol(I1,X,Y), sol(I2,X,Y), range(X;Y).
(7) :- prot(I1,A1), prot(I2,A2),  I2>1,
       I1 == I2-1, not next(I1,I2).
(8) next(I1,I2) :- prot(I1,A1), prot(I2,A2), I1<I2,
       sol(I1,X1,Y1), sol(I2,X2,Y2), range(X1;Y1;X2;Y2),
       1==abs(Y1-Y2)+abs(X2-X1).
(9) energy_pair(I1,I2) :- prot(I1,h), prot(I2,h),
       next(I1,I2), I1+2<I2, 1==(I2-I1) mod 2.
(10) maximize{ energy_pair(I1,I2) : prot(I1,h): prot(I2,h) }.
```

Rules `(1)` and `(2)`, together with the predicate `prot`, define the domains. Rule `(5)` implements the "generate" phase: it states that each aminoacid occupies exactly one position. Rules `(3)` and `(4)` fix the positions of the two initial aminoacids (they eliminates some symmetric solutions). The ASP-constraints `(6)` and `(7)` state that there are no self-loops and that two contiguous aminoacids must satisfy the `next` property. Rule `(8)` defines the `next` relation, also including the odd property of the lattice. The objective function is defined by Rule `(9)`, which determines the energy contribution of the aminoacids, and rule `(10)`, that searches for answer sets maximizing the energy. For the decision version of the problem, if `en` is the desired energy value, (10) is replaced by

```
(10')   en { energy_pair(I1,I2) : prot(I1,h): prot(I2,h) }.
```

**Results:** The experimental results for the two programs are reported in Table 5. The nature of this problem (prediction of *the* structure of a protein) justify us to test the systems on the optimization problem. Since CModels does not support optimization statements, we can only compare the performance of SICStus and SModels. Nevertheless, we performed a series of tests relative to the decision version of this problem, namely,

**Table 5.** Protein structure prediction ('–' denotes no answer within 30 hours of CPU-time)

| Instance | | | Optimization problem | | Decision problem | | |
|---|---|---|---|---|---|---|---|
| Input Sequence | Length | Min | CLP(FD) | SModels | CLP(FD) | SModels | CModels |
| $h^{10}$ | 10 | -4 | 0.13 | 0.74 | <0.01 | 0.53 | 1.01 |
| $h^{15}$ | 15 | -8 | 5.50 | 10.61 | 0.05 | 2.29 | 2.73 |
| $h^{20}$ | 20 | -12 | 766.22 | 1679.79 | 0.50 | 28.23 | 52.43 |
| $h^{25}$ | 25 | -16 | 103962.57 | – | 1664.35 | 2169.49 | 2620.94 |
| $(hpp)^3h$ | 10 | -4 | 0.10 | 0.51 | <0.01 | 0.35 | 0.33 |
| $(hpp)^5h$ | 16 | -6 | 0.22 | 22.33 | 0.08 | 16.06 | 15.94 |
| $(hpp)^7h$ | 22 | -8 | 46.87 | 1059.86 | 5.52 | 96.45 | 1609.75 |
| $(hpp)^9h$ | 28 | -10 | 14007.07 | – | 2815.62 | 2309.14 | 5813.23 |

answering the question "can the given protein fold to reach a given energy level?", using the energy results obtained by solving the optimization version of the problem. The results are also reported in Table 5.

## 7 Planning

Planning is one of the most interesting applications of ASP. CLP(FD) has been used less frequently to handle planning problems. A planning problem is based on the notions of `State` (a representation of the world) and `Actions` that change the states. We focus on solving a planning problem in the block world domain. Let us assume to have $N$ blocks (blocks $1, \ldots, N$). In the *initial state*, the blocks are arranged in a single stack, in increasing order, i.e., block 1 is on the table, block 2 is on top of block 1, etc. Block $N$ is on top of the stack. In the *goal state*, there must be two stacks, composed of the blocks with odd and even numbers, respectively. In both stacks the blocks are arranged in increasing order, i.e., blocks 1 and 2 are on the table and blocks $N-1$ and $N$ are on top of the respective stacks. The planning problem consists of finding a sequence of $T$ actions (*plan*) to reach the goal state, starting from the initial state. Some additional restrictions must be met: first, in each state at most three blocks can lie on the table. Moreover, a block $x$ cannot be placed on top of a block $y$ if $y \geq x$.

**CLP(FD):** We study the encoding of block world planning problem in CLP(FD). The code can be easily generalized as a scheme for encoding general planning problems. The plan can be modeled as a list `States` of $T+1$ states. Each `State` is a $N$-tuple `[B1,...,BN]`, where `Bi=j` means that block `i` is placed on block `j`. The case `j=0` represents the fact that the block `i` lies on the table. The initial state and the final state are represented by the lists $[0, 1, 2, 3, \ldots, \mathtt{N}-1]$ and $[0, 0, 1, 2, \ldots, \mathtt{N}-2]$.

```
planning(NBlocks,NTime) :- init_domains(NBlocks,NTime,States),
 initial_state(States), final_state(States),
 init_actions(NBlocks,NTime,Actions),
 forward(Actions,States), no_rep(Actions),
 action_properties(Actions,States),term_variables(Actions,Vars),
 labeling([leftmost],Vars).
init_domains(NBlocks,NTime,States) :- T1 is NTime+1,
 length(States,T1), init_domains(NBlocks,States).
init_domains(_,[]).
init_domains(N,[S|States]) :- length(S,N),
  init_domains(N,States), domain(S,0,N), count(0,S,'#=<',3).
initial_state([State|_]) :- increasing_list(State).
final_state(Sts) :- append(_,[[0|FS]],Sts), increasing_list(FS).
init_actions(_,0,[]) :- !.
init_actions(N,T,[[Block,To_Block]|Acts]) :- T1 is T-1,
  Block#\=To_Block, Block in 1..N, To_Block in 0..N,
  (Block#<To_Block #=> To_Block#=0), init_actions(N,T1,Acts).
forward([],_).
forward([[Block,To_Block]|B],[CurrState,NextState|Rest]) :-
  element(Block,NextState,To_Block), is_clear(CurrState,Block),
  is_clear(CurrState,To_Block), element(Block,CurrState,Old),
```

**Table 6.** Planning in blocks world ('–' denotes no answer in less than 3 hours)

| Instance Blocks | Instance Length | Plan exists | SModels | CModels | SICStus CLP(FD) | Instance Blocks | Instance Length | Plan exists | SModels | CModels | SICStus CLP(FD) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 12 | N | 0.29 | 0.12 | 0.01 | 7 | 50 | N | – | 542.98 | 586.73 |
| 5 | 13 | Y | 0.33 | 0.16 | 0.02 | 7 | 51 | N | – | 991.56 | 824.61 |
| 6 | 26 | N | 8.64 | 8.31 | 0.32 | 7 | 52 | N | – | 1091.54 | 1097.13 |
| 6 | 27 | Y | 12.17 | 6.56 | 0.26 | 7 | 53 | N | – | 2044.34 | 1509.35 |
| 7 | 42 | N | 355.66 | 220.00 | 42.83 | 7 | 54 | Y | – | 431.32 | 1104.16 |
| 7 | 43 | N | 565.60 | 74.19 | 58.91 | 8 | 56 | N | 3308.28 | 4667.86 | 3875.05 |
| 7 | 44 | N | 1126.52 | 169.01 | 80.59 | 8 | 57 | N | 4290.26 | 866.58 | 5101.24 |
| 7 | 45 | N | 2710.53 | 139.66 | 111.98 | 8 | 58 | N | 5672.42 | 287.16 | 7240.92 |
| 7 | 46 | N | 7477.13 | 299.01 | 158.03 | 8 | 59 | N | 7791.38 | 1769.51 | 9838.83 |
| 7 | 47 | N | – | 180.63 | 217.26 | 8 | 60 | N | 11079.03 | 903.10 | 13917.36 |
| 7 | 48 | N | – | 209.73 | 299.31 | 8 | 61 | N | 18376.59 | 488.78 | 19470.35 |
| 7 | 49 | N | – | 463.56 | 417.63 | 8 | 62 | N | 35835.76 | 4639.58 | 27030.19 |

```
  Old#\=To_Block, forward(B,[NextState|Rest]).
is_clear([],_).
is_clear([A|B],X) :- (X#\=0 #=> A#\=X), is_clear(B,X).
no_rep([_]).
no_rep([[X1,_],[X2,Y2]|Rest]):- X1#\=X2, no_rep([[X2,Y2]|Rest]).
action_properties([],_).
action_properties([[Block,_To]|Rest],[Current,Next|States]) :-
  inertia(1,Block,Current,Next),
  action_properties(Rest,[Next|States]).
inertia(_,_,[],[]).
inertia(N,X,[A|B],[C|D]) :-
  N1 is N+1, inertia(N1,X,B,D), (X#\=N #=> A#=C).
increasing_list(List) :- sequence(List,0).
sequence([],_).
sequence([N|R],N) :- M is N+1, sequence(R,M).
```

The code follows the usual constrain-and-generate methodology. The `init_domains` predicate generates the list of the `NTime` states and fixes the maximum number of objects admitted on the table in each state (using the built-in constraint `count`). After that, the initial and final states are initialized. The predicate `init_actions` specifies that a block can be moved either to the table or to another block having a smaller number. `forward` states that if a block is placed on another one, then both of them must be *clear*, i.e., without any block on top of them. The predicate `no_rep` guarantees that two consecutive actions cannot move the same block. Finally, `action_properties` forces the inertia laws (i.e., if a block is not moved, then it remains in its position).

**ASP:** There are several standard ways to encode a block world in ASP (e.g., [15,2]). The code used in our experiments is reported in [7].

**Results:** Table 6 reports the execution times from the three systems, for different number of blocks and plan lengths.

## 8 Knapsack

In this section we discuss a generalization of the knapsack problem. Let us assume to have $n$ types of objects, and each object of type $i$ has size $w_i$ and it costs $c_i$. We wish to fill a knapsack with $X_1$ objects of type 1, $X_2$ objects of type 2, and so on, so that:

$$\sum_{i=1}^{n} X_i w_i \leq \texttt{max\_size} \quad \text{and} \quad \sum_{i=1}^{n} X_i c_i \geq \texttt{min\_profit}. \tag{1}$$

where `max_size` is the capacity of the knapsack and `min_profit` is the minimum profit required.

**CLP(FD):** We represent the types of objects using two lists (containing the size and cost of each type of object). For instance, in our tests:

```
objects([2,4, 8,16,32,64,128,256,512,1024],
        [2,5,11,23,47,95,191,383,767,1535]).
```

The CLP(FD) encoding is:

```
knapsack(Max_Size,Min_Profit) :- objects(Weights,Costs),
       length(Sizes,N), length(Vars,N), domain(Vars,0,Max_Size),
       scalar_product(Sizes,Vars,#=<,Max_Size),
       scalar_product(Costs,Vars,#>=,Min_Profit),
       labeling([ff],Vars).
```

Observe that we used the built-in predicate `scalar_product` for implementing (1). The built-in predicate `knapsack`, available in SICStus Prolog, is a special case of `scalar_product` where the third argument is the equality constraint.

**ASP:** Input representation is given by facts of the form: `item(Item,Weight,Cost).`

```
item(1,2,2).    item(2,4,5).   item(3,8,11).    item(4,16,23).
item(5,32,47).  item(6,64,95). item(7,128,191). item(8,256,383).
item(9,512,767).               item(10,1024,1535).
```

The knapsack problem can be encoded as follows:

```
(1)  occs(0..max_size).
(2)  item_occs(I,Item_Occurences,W,C) :-
        item(I,W,C), occs(O), Item_Occurences = O/W.
(3)  1{in_sack(I,IO,W,C):item_occs(I,IO,W,C)}1 :- item(I,W,C).
(4)  cond_cost :-
      min_profit [in_sack(I,IO,W,C):item_occs(I,IO,W,C) = IO*C].
(5)  :- not cond_cost.
(6)  cond_weight :-
       [in_sack(I,IO,W,C):item_occs(I,IO,W,C) = IO*W] max_size.
(7)  :- not cond_weight.
```

Fact `(1)` fixes the domain for the occurrences of items in the knapsack. Rule `(2)`, instead, fixes the possible occurrences for each item in the knapsack. Rule `(3)` states that, for each type of objects `I`, there is only one fact `in_sack(I,IO,W,C)` in the answer set, representing the number of objects of type `I` in the knapsack. This trick is not needed in CLP(FD), where the same effect is obtained by bounds consistency. Rules `(4)`–`(7)` establish the constraints of minimum profit and maximum size. The two constants `max_size` and `min_profit` must be provided to lparse during grounding.

**Table 7.** Knapsack instances ('–' denotes no answer within 30 minutes of CPU-time)

| max_size | min_profit | Answer | SICStus | SModels |
|---|---|---|---|---|
| 255 | 374 | Y | 0.02 | 0.04 |
| 255 | 375 | N | 0.03 | 3.08 |
| 511 | 757 | Y | 0.36 | 0.12 |
| 511 | 758 | N | 0.36 | 130.82 |
| 1023 | 1524 | Y | 8.81 | 0.49 |
| 1023 | 1525 | N | 8.75 | – |
| 2047 | 3059 | Y | 368.50 | 1.84 |
| 2047 | 3060 | N | 366.79 | – |

**Results:** Table 7 reports some of the results we obtained. CModels seems unable to properly deal with this problem: for any of the instances we experimented with (except the smallest ones, involving at most five types of objects) the corresponding process was terminated by the operative system. The reason for this could be found by observing that the run-time images of such processes grow very large in size (up to 4.5GB, in some instances). We have also encoded this problem using the `weight` declarations, but the presented code is faster and less sensible to size of the numbers.

## 9 Discussion and Conclusions

We tested the CLP(FD) and ASP codes for various combinatorial problems. In the Tables
1–7 we reported the running times (in seconds) of the solutions to these problems on different problem instances. Let us try here to analyze these results.

First of all, from the benchmarks, it is clear that ASP provides a more compact, and probably more declarative, encoding; in particular, the reliance on grounding and domain-restricted variables allows ASP to avoid the use of recursion in many situations.

As far as running times are concerned, CLP(FD) definitely wins the comparison vs. SModels. In a few cases, the running times are comparable, but in most of the cases CLP(FD) runs significantly faster. Observe also that CModels is, in most of the problems, faster than SModels; part of this can be justified by the fact that the programs we are using are mostly tight [9], and by the high speed of the underlying SAT solver used by CModels.

The comparison between CLP(FD) and CModels is more interesting. In the $k$-coloring and $N$-$M$-queens cases, running times are comparable. In some of the classes of graphs, CModels performs slightly better on all instances. More in general, whenever the instances of a single class are considered, one of the two systems tends to always outperform the other. This indicates that the behavior of the solver is significantly affected by the nature of the specific problem instances considered (recall that each class of graphs comes from encodings of instances of different problems [24]).

As one may expect, the bottom-up search strategy of ASP is less sensitive to the presence of solutions w.r.t. the top down search strategy of CLP(FD). As a matter of fact, CLP(FD) typically runs faster than CModels when a solution exists. Moreover, CLP(FD) behaves better on small graphs. For the Hamiltonian circuit problem, CLP(FD) runs significantly faster—we believe this is due to the use of the built-in global constraint `circuit`, which guarantees excellent constraint propagation. In this

**Table 8.** Schematic results' analysis. + (-) means that the formalism is (not) applicable. ++ that it is the best when the two formalisms are applicable.

| | Coloring | Hamilton | Schur | PF | Planning | Knapsack |
|---|---|---|---|---|---|---|
| CLP(FD) | + | ++ | + | + | + | + |
| ASP CModels | ++ | + | + | - | + | - |

case, only in absence of solutions the running times are comparable—i.e., when the two approaches are forced to traverse the complete search tree. A similar situation arises in computing Schur numbers. When the solution exists CLP(FD) performs better. On the other hand, whenever there is no solution, running times are favorable to CModels.

Regarding the protein folding problem, CLP(FD) solves the optimization problems much faster than ASP. In the decision version, times are closer. Also in this case, however, the ASP code appears to be simpler and more compact than the CLP(FD) one. In general, in designing the CLP code, the programmer cannot easily ignore knowledge about the inference strategy implemented in the CLP engine. The fact that CLP(FD) adopts a top-down depth-first strategy influences programmer's choices in encoding the algorithms.

For the planning problem, we observe that SModels runs faster than CModels for small instances. In general, CLP(FD) performs better for small dimensions of the problem. On the other hand, when the dimension of the problem instance becomes large, the behavior of CLP(FD) and SModels become comparable while CModels provides the best performance. In fact, the performance of CModels does not seem to be significantly affected by the growth in the size of the problem instance, as clearly happens for CLP(FD) and SModels. The same phenomenon can be also observed in other situations, e.g., in the Hamiltonian circuit and Schur numbers problems. In these cases, the time spent by CModels to obtain a solution does not appear to be directly related to the raw dimension of the problem instance. Initial experiments reveal that this phenomenon arises even when different SAT-solvers are employed. Further studies are needed to better understand to which extent the intrinsic structure of an instance biases CModels' behavior, in particular the way in which CModels' engine translates an ASP program into a SAT-instance.

For the Knapsack problem, CModels is not applicable. CLP(FD) runs definitively faster than SModels; furthermore, SModels becomes inapplicable for large problem instances.

Table 9 intuitively summarizes our observations drawn from the different benchmarks. Although these experiments are quite preliminary, they already provide some concrete indications that can be taken into account when choosing a paradigm to tackle a problem. We can summarize the main points as follows:

- graph-based problems have nice compact encodings in ASP and the performance of the ASP solutions is acceptable and scalable;
- problems requiring more intense use of arithmetic and/or numbers are declaratively and efficiently handled by CLP(FD);
- for problems with no arithmetic, the exponential growth w.r.t. the input size is less of an issue for ASP.

A comparison between CLP(FD) implementations is outside the scope of this paper (see, e.g., [10,20,6]). Nevertheless, we tested the CLP(FD) programs using B-Prolog, ECLiPSe, and GNU Prolog [23]. As far as the running times are concerned, these experiments indicated that B-Prolog and SICStus Prolog have comparable behavior, GNU Prolog is the fastest, and ECLiPSe the slowest.

We have excluded the grounding phase from the ASP timings. In our tests it is negligible w.r.t. the SModels/CModels running time, save for easier instances (with running times shorter than one second). Moreover, we also tested CModels with other SAT solvers (SIMO, RELSAT, ZCHAFF). As one can expect (see, e.g., [12]) the choice of the SAT solver influences performance, but there is no clear winner.

In the future we plan to extend our analysis to other problems and to other constraint solvers (e.g., ILOG) and ASP-solvers (e.g., ASSAT, aspps, DLV). In particular, we are interested in answering the following questions:

- is it possible to formalize domain and problem characteristics to lead the choice of which paradigm to use?
- is it possible to introduce strategies to split problem components and map them to cooperating solvers (using the best solver for each part of the problem)?

In particular, we are interested in identifying those contexts where the ASP solvers perform significantly better than CLP. It seems reasonable to expect this behavior, for instance, whenever incomplete information comes into play.

*Acknowledgments.* We thank the anonymous referees for their patience and for their useful suggestions; in particular for the ASP encoding of the Knapsack problem. This work is partially supported by the GNCS2005 project on constraints and their applications and NSF grants CNS-0220590, HRD-0420407, and CNS-0454066.

## References

1. C. Anger, T. Schauß, and M. Truszczyński. ASPARAGUS – the Dagstuhl Initiative. *ALP Newsletter*, 17(3), 2004.
2. C. Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003.
3. P. Clote and R. Backofen. *Computational Molecular Biology*. Wiley & Sons, 2001.
4. P. Crescenzi et al. On the complexity of protein folding. In *STOC*, pages 597–603, 1998.
5. A. Dal Palù, A. Dovier, and F. Fogolari. Constraint logic programming approach to protein structure prediction. *BMC Bioinformatics*, 5(186):1–12, 2004.
6. D. Diaz and P. Codognet. Design and Implementation of the GNU Prolog System. *Journal of Functional and Logic Programming* 2001(6), 2001.
7. A. Dovier, A. Formisano, and E. Pontelli. A comparison of CLP(FD) and ASP solutions to combinatorial problems. `www.di.univaq.it/~formisano/CLPASP`
8. I. Elkabani, E. Pontelli, and T. C. Son. SModels with CLP and Its Applications: A Simple and Effective Approach to Aggregates in ASP. In *ICLP*, 73–89, 2004.
9. E. Erdem and V. Lifschitz. Tight Logic Programs. In *TPLP*, 3:499-518, 2003.
10. A. J. Fernandez and P. M. Hill. A Comparative Study of 8 Constraint Programming Languages Over the Boolean and Finite Domains. *Constraints*, 5(3):275–301, 2000.
11. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *ICLP*, pages 1070–1080, MIT Press, 1988.

12. E. Giunchiglia, Y. Lierler, and M. Maratea. SAT-Based Answer Set Programming. In Proc. of *AAAI'04*, pages 61–66, AAAI/Mit Press, 2004.
13. J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *J. of Logic Programming*, 19/20:503–581, 1994.
14. Y. Lierler and M. Maratea. CModels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In *LPNMR*, pages 346–350. Springer Verlag, 2004.
15. V. Lifschitz. Answer Set Planning. In *Logic Programming and Non-monotonic Reasoning*, pages 373–374. Springer Verlag, 1999.
16. F. Lin and Y. Zhao. ASSAT: Computing Answer Sets of A Logic Program By SAT Solvers. In Proc. of *AAAI'02*, pages 112–117. AAAI/MIT Press, 2002.
17. V. W. Marek and M. Truszczyński. Autoepistemic Logic. *JACM*, 38(3):588–619, 1991.
18. V. W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. *The Logic Programming Paradigm*, 375–398. Springer, 1999.
19. K. Marriott and P. J. Stuckey. *Programming with Constraints*. MIT Press, 1998.
20. M. Wallace, J. Schimpf, K. Shen, and W. Harvey. On Benchmarking Constraint Logic Programming Platforms. *Constraints*, 9(1):5–34, 2004.
21. E. W. Weisstein. *Schur Number*. From MathWorld–A Wolfram Web Resource. `mathworld.wolfram.com/SchurNumber.html`.
22. Web references for some ASP solvers. ASSAT: `assat.cs.ust.hk`. CCalc: `www.cs.utexas.edu/users/tag/cc`. CModels: `www.cs.utexas.edu/users/tag/cmodels`. DeReS and aspps: `www.cs.uky.edu/ai`. DLV: `www.dbai.tuwien.ac.at/proj/dlv`. SModels: `www.tcs.hut.fi/Software/smodels`.
23. Web references for some CLP(FD) implementations. SICStus Prolog: `www.sics.se/isl/sicstuswww/site/index.html`. B-Prolog: `www.probp.com`. ECLiPSe: `www.icparc.ic.ac.uk/eclipse`. GNU Prolog: `pauillac.inria.fr/~diaz/gnu-prolog`.
24. Web site of COLOR02/03/04: Graph Coloring and its Applications: `http://mat.gsia.cmu.edu/COLORING03`.