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))* [16]. 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 [6]).

It is well-known [14, 2] that, given a propositional normal logic program P, deciding whether or not it admits an *answer set* [9] is a 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 [19] 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 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* [10, 16]. In this context, a finite

* This work is partially supported by GNCS2005 project on constraints and their applications.

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 [16].

In this paper, we report the outcomes of 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 widely 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 in favor of potential cross-fertilizations. This study also complements the system benchmarking studies, that have recently appeared for both CLP(FD) systems [8, 17] and ASP solvers [1, 13, 11].

2 The Experimental Framework

In order to conduct our experiments, we selected 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., ECLiPSe). The choice of SICStus has been suggested by its good performance (better than ECLiPSe on some of the benchmarks at hand). 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 [19]. 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 time needed for computing the first solution, if any (CPU usage time)—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 are reported in `www.di.univaq.it/~formisano/CLPASP`.

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" [20], 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 (a list of integers), while the second argument is the set 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(Output, Edges), labeling([ff], Colors).
create_output([],[],[]).
create_output([N | Nodes], [C|Colors], [N-C|Output]) :-
      create_output(Nodes, Colors, Output).
different(_, []).
different(Output, [[A,B]|R]) :- member(A-CA, Output),
      member(B-CB, Output), CA #\= CB, different(Output, R).
```

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` options 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. Nodes are represented, as before, by natural numbers. Edges are rendered by facts, as in the following instance:

```
node(1..138). edge(1,36). edge(2,45). ... edge(138,7). 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 (the parameter is a constant to be initialized in 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).

Results: We tested the above programs on more than one hundred instances drawn from [20]. 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

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	–	
2-Insertions_5	597 × 3936	N	45.08	0.50	6.97	?	–	–	–	?	–	–	–	
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	
DSJR500.1c	500 × 121275	N	84.19	3.66	6.07	N	1083.17	4.54	6.18	N	9881.35	5.50	6.19	
DSJC500.9	500 × 224874	N	74.44	3.39	5.67	N	543.02	4.29	5.67	N	3146.96	5.09	5.77	
DSJC1000.1	1000 × 49629	N	12.99	1.61	5.01	N	241.43	2.02	5.06	N	–	3.61	–	
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	
gen400_p0.9_75	400 × 71820	N	38.87	2.17	2.88	N	270.12	2.70	2.89	N	1608.19	3.22	2.94	
inithx.i.1	864 × 18707	N	4.92	0.65	2.28	N	57.15	0.81	2.29	N	415.41	1.00	2.32	
inithx.i.2	645 × 13979	N	3.50	0.50	1.28	N	34.19	0.63	1.28	N	268.87	0.83	1.31	
inithx.i.3	621 × 13969	N	3.50	0.50	1.22	N	34.14	0.64	1.24	N	268.36	0.80	1.26	
le450_5a	450 × 5714	N	0.85	0.24	0.26	N	9.06	0.29	0.28	Y	190.38	12.30	5.29	
le450_5b	450 × 5734	N	0.85	0.24	0.29	N	7.77	0.29	0.3	Y	5146.86	0.98	0.48	
le450_5c	450 × 9803	N	1.64	0.37	0.44	N	9.98	0.46	0.44	Y	217.77	0.70	0.03	
le450_5d	450 × 9757	N	1.64	0.36	0.44	N	10.29	0.44	0.47	Y	530.63	0.60	0.08	
mulsol.i.1	197 × 3925	N	0.58	0.17	0.10	N	2.80	0.20	0.10	N	19.07	0.24	0.11	
mulsol.i.2	188 × 3885	N	0.57	0.17	0.10	N	2.83	0.20	0.09	N	31.25	0.24	0.11	
mulsol.i.3	184 × 3916	N	0.58	0.16	0.09	N	2.86	0.20	0.10	N	31.93	0.25	0.10	
mulsol.i.4	185 × 3946	N	0.58	0.17	0.10	N	2.92	0.20	0.10	N	32.83	0.25	0.11	
mulsol.i.5	186 × 3973	N	0.59	0.17	0.10	N	2.93	0.20	0.09	N	33.02	0.26	0.11	
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 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.)

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 [20] 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 $N \times N$ chessboard. 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 or not 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 run 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.

Instance		Solvability for $M = N - 1$					Solvability for $M = N$					Solvability for $M = N + 1$				
N	$V \times E$		SModels	CModels	CLP(FD)	ugraph		SModels	CModels	CLP(FD)	ugraph		SModels	CModels	CLP(FD)	ugraph
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	–	?	–	–	–	–	?	–	–	–	–

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

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.

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

```
hc(N, Edges) :- 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 [15]:

```
(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).
```

The description of the search space is given by rules `(1)` and `(2)`. They state that, 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`).

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
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	–
nv50a260	50×239	Y	0.12	0.95	0.01
nv50a280	50×263	Y	0.13	0.51	0.02
nv50a300	50×280	Y	0.14	0.16	0.02
nv50a340	50×303	Y	0.14	1.25	0.03
nv50a360	50×329	Y	0.15	1.14	0.02
nv50a380	50×354	Y	0.15	0.62	0.03
nv50a400	50×365	Y	0.16	0.17	0.02
nv50a420	50×375	Y	0.15	0.18	0.01

Instance	node×edges	Hamiltonian?			
			SModels	CModels	CLP(FD)
nv50a440	50×401	Y	0.16	3.08	0.01
nv50a460	50×416	Y	0.17	0.22	0.02
nv50a480	50×422	Y	0.17	3.03	0.02
nv50a500	50×438	Y	0.17	0.71	0.03
nv50a520	50×459	Y	0.18	0.70	0.03
nv50a540	50×480	Y	0.18	1.18	0.01
nv50a560	50×500	Y	0.19	0.26	0.02
nv50a580	50×509	Y	0.18	0.42	0.03
nv60a320	60×304	Y	0.19	0.79	0.04
nv60a360	60×343	Y	0.19	8.15	0.03
nv60a420	60×389	Y	0.21	0.96	0.03
nv60a440	60×412	Y	0.23	8.90	0.03
nv60a460	60×423	Y	0.22	0.84	0.04
nv60a480	60×425	Y	0.22	1.60	0.03
nv60a500	60×455	Y	0.23	2.18	0.04
nv60a520	60×582	Y	0.23	0.35	0.03
nv60a540	60×587	Y	0.24	0.55	0.02
nv60a560	60×513	Y	0.26	0.20	0.03
nv60a580	60×532	Y	0.25	0.99	0.04
nv70a300	70×287	Y	0.21	0.79	0.04
nv70a320	70×306	Y	0.23	4.24	0.04
nv70a340	70×328	Y	0.24	1.87	0.05
nv70a360	70×346	Y	0.24	1.44	0.02
nv70a380	70×359	Y	0.25	0.44	0.04
nv70a400	70×386	Y	0.26	4.22	0.04
nv70a420	70×404	Y	0.27	4.63	0.04
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

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

Results: Most of the problem instances have been taken from the benchmarks used to compare ASP-solvers [13]. 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 graphs 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 [19]. 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 interval $[1..n]$ 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 $160 \leq S(5) \leq 315$ [18].

CLP(FD): The following CLP(FD) code checks if N can be partitioned into P sum-free parts, i.e., if $S(P) \geq N$.

```
schur(N,P) :- length(List,N), domain(List,1,P),
              constraints(List,N), labeling([ff],List).
constraints(List, N) :- recursion(List,1,1,N).
recursion( _  ,I,_,N):- I>N, !.
```

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)
$\langle 3, 11\rangle$	Y	0.01	0.04	<0.01	$\langle 5, 100\rangle$	Y	–	0.39	0.31
$\langle 3, 12\rangle$	Y	0.01	0.04	<0.01	$\langle 5, 101\rangle$	Y	–	0.43	0.29
$\langle 3, 13\rangle$	Y	0.01	0.04	<0.01	$\langle 5, 102\rangle$	Y	–	0.41	0.35
$\langle 3, 14\rangle$	N	0.02	0.04	0.01	$\langle 5, 103\rangle$	Y	–	0.74	0.36
$\langle 3, 15\rangle$	N	0.02	0.04	0.03	$\langle 5, 104\rangle$	Y	–	5.01	0.34
$\langle 3, 16\rangle$	N	0.02	0.04	0.03	$\langle 5, 105\rangle$	Y	–	27.88	0.37
$\langle 4, 40\rangle$	Y	0.22	0.10	0.30	$\langle 5, 106\rangle$	Y	–	38.55	0.40
$\langle 4, 41\rangle$	Y	0.24	0.23	1.17	$\langle 5, 107\rangle$	Y	–	5.07	0.44
$\langle 4, 42\rangle$	Y	0.25	0.21	2.61	$\langle 5, 108\rangle$	Y	–	2.80	0.43
$\langle 4, 43\rangle$	Y	0.27	0.24	2.51	$\langle 5, 109\rangle$	Y	–	13.97	0.44
$\langle 4, 44\rangle$	Y	0.29	3.35	4.18	$\langle 5, 110\rangle$	Y	–	33.12	54.71
$\langle 4, 45\rangle$	N	510.01	891.66	–	$\langle 5, 111\rangle$	Y	–	0.52	56.72
$\langle 4, 46\rangle$	N	561.80	813.34	–	$\langle 5, 112\rangle$	Y	–	0.54	58.44
$\langle 4, 47\rangle$	N	767.80	791.57	–	$\langle 5, 113\rangle$	Y	–	0.54	207.46
$\langle 4, 48\rangle$	N	978.84	805.33	–	$\langle 5, 114\rangle$	Y	–	11.63	1032.43
$\langle 4, 49\rangle$	N	1258.57	678.19	–	$\langle 5, 115\rangle$	Y	–	82.13	1069.54
$\langle 4, 50\rangle$	N	1680.34	890.05	–	$\langle 5, 116\rangle$	Y	–	60.53	1108.02
$\langle 4, 51\rangle$	N	1892.36	1046.73	–	$\langle 5, 117\rangle$	Y	–	761.11	1150.25

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

```
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).
```

The term `List` is a list of `N` variables (associated to the numbers `1`,...,`N`), each of them taking values in `1..P`. The value of the ith variable identifies the block of the partition i belongs to. The predicate `recursion` states that for all `I` and `J`, with $\texttt{1} \leq \texttt{I} \leq \texttt{J} \leq \texttt{N}$, the numbers `I`, `J` and `I+J` must not be all in the same block.

ASP: The following is the ASP solution we employed.

```
(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).
```

The atom `inpart(X,P)` represents the fact that number X is assigned to part P. Rule `(2)` generates the potential solutions, by assigning each integer to exactly one part. The ASP-constraints `(3)`, `(4)`, and `(5)` remove unwanted solutions: `(3)` states that for any X and Y, the three numbers X, Y, and $X + Y$ cannot belong to the same part. Rules `(4)` and `(5)` remove symmetries, by selecting the free part with lowest index.

Results: Table 4 lists the timings 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

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

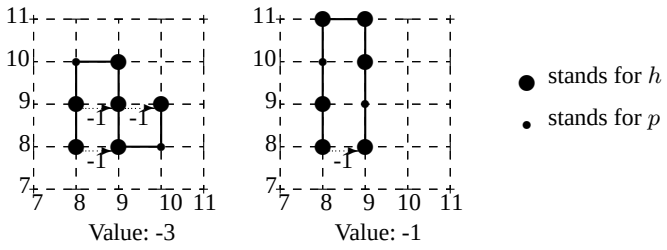

Fig. 1. Two foldings for $S = hhphhhph$ ($n = 8$). The leftmost one is minimal.

and minimizing the energy:

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

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 $|X_1 - X_2| + |Y_1 - Y_2| = 1$. W.l.o.g., we set $\omega(1) = \langle n, n\rangle$ and $\omega(2) = \langle n, n+1\rangle$, to remove some symmetries in the solution space.

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 can be found in `www.di.univaq.it/~formisano/CLPASP`. 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].

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 $(\mathsf{hpp})^3\mathsf{h}$ using regular expressions) is described as:

```
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),neq(I1,I2),
       sol(I1,X,Y), sol(I2,X,Y), range(X;Y).
(7) :- prot(I1,A1), prot(I2,A2), I2>1, eq(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).
```

Instance			Optimization problem		Decision problem		
Input Sequence	Length	Min	CLP(FD)	SModels	CLP(FD)	SModels	CModels
h^{10}	10	-4	0.14	0.91	0.01	0.65	0.69
h^{15}	15	-8	1.93	13.21	0.04	2.84	2.69
h^{20}	20	-12	201.58	1982.44	0.29	45.63	40.70
h^{25}	25	-16	25576.38	–	817.71	3181.78	1165.26
$(hpp)^3h$	10	-4	0.01	0.66	0.01	0.42	0.49
$(hpp)^5h$	16	-6	0.32	26.95	0.22	22.46	16.58
$(hpp)^7h$	22	-6	62.69	1303.13	12.75	35.96	161.25
$(hpp)^9h$	28	-9	6758.45	–	1955.28	3369.78	1217.34

Table 5. Protein structure prediction ('–' denotes no answer within 10 hours of CPU-time).

```
(9) energy_pair(I1,I2) :- prot(I1,h), prot(I2,h),
       next(I1,I2), I1+2<I2, 1==(I2-I1) mod 2.
(10) seq_proteins(I1,I2) :- prot(I1,A1), prot(I2,A2),
       I1+2<I2, 1==(I2-I1) mod 2.
(11) maximize{ energy_pair(I1,I2) : seq_proteins(I1,I2) }.
```

Rules `(1)` and `(2)`, together with the predicate `prot`, define the domains. The range $N-\sqrt{N}..N+\sqrt{N}$ is a heuristic value used consistently in both the CLP(FD) and ASP encodings. 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 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 `(11)`, that searches for a answer sets maximizing the energy.

Results: The experimental results for the two programs are reported in Table 5. 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, 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),
  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

Instance		Plan	SModels	CModels	SICStus	Instance		Plan	SModels	CModels	SICStus
Blocks	Length	exists			CLP(FD)	Blocks	Length	exists			CLP(FD)
5	11	N	0.23	0.11	0.01	7	51	N	–	991.56	824.61
5	12	N	0.29	0.12	0.01	7	52	N	–	1091.54	1097.13
5	13	Y	0.33	0.16	0.02	7	53	N	–	2044.34	1509.35
6	22	N	2.61	8.16	0.11	7	54	Y	–	431.32	1104.16
6	23	N	3.60	9.86	0.13	8	40	N	193.31	115.40	21.73
6	24	N	4.73	6.46	0.18	8	41	N	234.96	300.55	29.48
6	25	N	6.44	13.40	0.25	8	42	N	279.35	126.93	41.57
6	26	N	8.64	8.31	0.32	8	43	N	335.08	196.62	55.66
6	27	Y	12.17	6.56	0.26	8	44	N	404.43	874.70	78.75
7	33	N	38.64	175.96	2.49	8	45	N	475.71	231.80	110.08
7	34	N	47.34	222.07	3.42	8	46	N	579.54	351.47	158.71
7	35	N	58.01	153.02	4.71	8	47	N	682.70	193.02	205.57
7	36	N	71.40	106.57	6.36	8	48	N	808.11	52.04	285.94
7	37	N	87.84	115.96	8.70	8	49	N	947.37	463.65	386.23
7	38	N	107.11	157.32	11.94	8	50	N	1123.94	379.32	544.91
7	39	N	150.11	84.98	16.33	8	51	N	1328.18	192.87	748.38
7	40	N	177.69	115.40	22.22	8	52	N	1566.62	172.24	1049.58
7	41	N	253.65	217.10	31.19	8	53	N	1877.88	3440.71	1436.14
7	42	N	355.66	220.00	42.83	8	54	N	2257.87	212.60	2028.70
7	43	N	565.60	74.19	58.91	8	55	N	2717.22	178.01	2760.98
7	44	N	1126.52	169.01	80.59	8	56	N	3308.28	4667.86	3875.05
7	45	N	2710.53	139.66	111.98	8	57	N	4290.26	866.58	5101.24
7	46	N	7477.13	299.01	158.03	8	58	N	5672.42	287.16	7240.92
7	47	N	–	180.63	217.26	8	59	N	7791.38	1769.51	9838.83
7	48	N	–	209.73	299.31	8	60	N	11079.03	903.10	13917.36
7	49	N	–	463.56	417.63	8	61	N	18376.59	488.78	19470.35
7	50	N	–	542.98	586.73	8	62	N	35835.76	4639.58	27030.19

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

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 ways to encode a block world in ASP (e.g., [12, 2]). In our experiments we adopted the code reported in `www.di.univaq.it/~formisano/CLPASP`.

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 object 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), e.g., `objects([2,4,8,16,32,64], [2,5,11,23,47,95])`. 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 `scalar_product` for implementing (1).[4]

ASP: The ASP-formulation of the knapsack problem we experimented with, is easily obtainable from the classical encoding of the standard knapsack problem (i.e., with $X_i \in \{0, 1\}$). The different kinds of objects are so represented as follows:

```
item(1..10).
#weight size(1,X1) = 2*X1.     #weight size(2,X2) = 4*X2.
#weight size(3,X3) = 8*X3.     #weight size(4,X4) = 16*X4.
#weight size(5,X5) = 32*X5.    #weight size(6,X6) = 64*X6.
#weight size(7,X7) = 128*X7.   #weight size(8,X8) = 256*X8.
#weight size(9,X9) = 512*X9.   #weight size(10,X10) = 1024*X10.
#weight cost(1,X1) = 2*X1.    #weight cost(2,X2) = 5*X2.
#weight cost(3,X3) = 11*X3.   #weight cost(4,X4) = 23*X4.
#weight cost(5,X5) = 47*X5.   #weight cost(6,X6) = 95*X6.
#weight cost(7,X7) = 191*X7.  #weight cost(8,X8) = 383*X8.
#weight cost(9,X9) = 767*X9.  #weight cost(10,X10) = 1535*X10.
```

The ASP encoding is the following:

```
(1) occs(0..max_size).
(2) 1 { in_sack(I,XI) : occs(XI) } 1 :- item(I).
(3) size(I,XI) :- item(I), occs(XI), in_sack(I,XI).
(4) cost(I,XI) :- item(I), occs(XI), in_sack(I,XI).
(5) cond_cost :- min_profit [ cost(I,XI) : item(I) : occs(XI) ].
(6) :- not cond_cost.
(7) cond_weight :- [ size(I,XI) : item(I) : occs(XI) ] max_size.
(8) :- not cond_weight.
```

Fact `(1)` fixes the domain for the variable `XI`. The Rule `(2)` states that, for each type of objects `I`, there is only one fact `in_sack(I,XI)` in the answer set, representing the number of objects of type `I` in the knapsack. Rules `(3)` and `(4)` get the total `size` and `cost` for each type of object present in the knapsack. Rules `(5)`–`(8)` establish the constraints of minimum profit and maximum size. The two constants `max_size` and `min_profit` must be provided to lparse during grounding.

Results: Table 7 reports some of the results we obtained. The right-hand side columns regard runs with 10-fold increase in objects' costs and `min_profit`. We were not able to obtain any result from CModels. 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). The mark (*) in Table 7, denotes instances where SModels is not able to process the ground program—in such cases, SModels stops with the message "`sum of weights in weight rule too large...`". The (**) denotes

[4] The built-in predicate `knapsack`, available in SICStus Prolog, is a special case of `scalar_product` where the third argument is the equality constraint.

max_size	min_profit	Answer	SICStus	SModels	max_size	min_profit	Answer	SICStus	SModels
255	374	Y	0.02	0.34	255	3740	Y	0.02	0.35
255	375	N	0.03	6.25	255	3750	N	0.03	6.27
511	757	Y	0.36	0.85	511	7570	Y	0.36	(**)
511	758	N	0.36	248.26	511	7580	N	0.36	0.50
1023	1524	Y	8.81	2.59	1023	15240	Y	8.75	(**)
1023	1525	N	8.75	–	1023	15250	N	8.69	1.03
2047	3059	Y	368.50	(*)	2047	30590	Y	369.83	(**)
2047	3060	N	366.79	(*)	2047	30600	N	368.24	1.83

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

instances where SModels reports the incorrect answer (No). For $(511, 7570)$, setting the value `#weight cost(10,Q) = 1177*Q` we obtain the correct solution (that does not use objects of type 10), while with values greater than 1177 something goes astray—probably improperly handled large integers. This shows that currently ASP is more sensible to number size w.r.t. CLP(FD). The behavior of SModels on $(511, 7580)$, $(1023, 15250)$, and $(2047, 30600)$ should be wrong, as well, but the absence of solutions does not allow to point out it.

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 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 [7], 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 [20]).

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 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 and numbers are low, CLP(FD)

performs better. For larger instances (even with solutions), the running times are favorable to CModels.

Regarding the protein folding problem, CLP(FD) solves the problems much faster than ASP. 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 and unreliable for large problem instances.

Table 9 intuitively summarizes our observations drawn from the different benchmarks. Although these experiments are quite preliminary, they actually provide already 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.

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

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

In the future we plan to extend our analysis to other problems and to other constraint solvers (e.g., BProlog, 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.

References

[1] C. Anger, T. Schaub, and M. Truszczynski. 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] 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.

[7] E. Erdem and V. Lifschitz. Tight Logic Programs. In *TPLP*, 3:499-518, 2003.

[8] 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.

[9] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *ICLP*, pages 1070–1080, MIT Press, 1988.

[10] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *J. of Logic Programming*, 19/20:503–581, 1994.

[11] 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.

[12] V. Lifschitz. Answer Set Planning. In *Logic Programming and Non-monotonic Reasoning*, pages 373–374. Springer Verlag, 1999.

[13] F. Lin and Y. Zhao. ASSAT: Computing Answer Sets of A Logic Program By SAT Solvers. In *AAAI*, pages 112–117. AAAI/MIT Press, 2002.

[14] V. W. Marek and M. Truszczynski. Autoepistemic Logic. *JACM*, 38(3):588–619, 1991.

[15] V. W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. *The Logic Programming Paradigm*, 375–398. Springer, 1999.

[16] K. Marriott and P. J. Stuckey. *Programming with Constraints*. MIT Press, 1998.

[17] M. Wallace, J. Schimpf, K. Shen, and W. Harvey. On Benchmarking Constraint Logic Programming Platforms. *Constraints*, 9(1):5–34, 2004.

[18] E. W. Weisstein. *Schur Number*. From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/SchurNumber.html`.

[19] 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`.

[20] Web site of COLOR02/03/04: Graph Coloring and its Applications: `http://mat.gsia.cmu.edu/COLORING03`.