An Empirical Study of Constraint Logic Programming and Answer Set Programming Solutions of Combinatorial Problems*

Agostino Dovier† Andrea Formisano‡ Enrico Pontelli§

Abstract

This paper presents experimental comparisons between the declarative encodings of various computationally hard problems in both *Answer Set Programming (ASP)* and *Constraint Logic Programming over finite domains (CLP(FD))*. The objective is to investigate 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 foundations for transfer of technology between the two domains, e.g., suggesting ways to use CLP(FD) in the execution of ASP.

Keywords. *D*eclarative Programming, Constraint Logic Programming, Answer Set Programming, Solvers, Benchmarking.

1 Introduction

The objective of this work is to experimentally compare the use of two distinct logic-based paradigms, traditionally recognized as excellent tools to tackle computationally hard problems. The two paradigms considered are *Answer Set Programming (ASP)* [2] and *Constraint Logic Programming over Finite Domains (CLP(FD))* [27]. 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 indicate methods for integration and cooperation between the two paradigms (e.g., along the lines of [10, 11]).

*A preliminary version of this paper appeared in the Proceedings of the International Conference on Logic Programming, LNCS 3668, pp. 67–82, Springer Verlag, 2005.

†Univ. di Udine, Dip. di Matematica e Informatica. dovier@dimi.uniud.it

‡Univ. di L'Aquila, Dip. di Informatica. formisano@di.univaq.it

§New Mexico State University, Dept. Computer Science. epontell@cs.nmsu.edu

It is well-known [2, 25] that, given a propositional normal logic program P, deciding whether P admits an *answer set* [15] 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 [36] are programs designed to compute 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 search space. Most ASP solvers rely on variations of the Davis-Putnam-Logemann-Loveland procedure [6, 7] in their computations. Such solvers are often equipped with a front-end that transforms a collection of non-propositional normal logic programming clauses (with limited use of function symbols) to a *finite* set of ground instances of such clauses. Some solvers provide also syntactic extensions to facilitate program development—e.g., limited forms of aggregation [31]—and 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* [19, 27]. In this context, a finite domain of objects (typically integers) is associated to each variable in the problem specification, and the constraints are literals of the forms $s = t$, $s \neq t$, $s < t$, $s \leq t$, etc., where s and t are arithmetic expressions. This type of framework supports the natural and declarative encoding of search strategies and NP-complete problems. Indeed, a rich literature has been developed presenting applications of CLP(FD) to a variety of search and optimization problems [27].

In this paper, we report the outcome of a study 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 framework, attempting to encode the various problems in the *most declarative* way possible. In particular, we adopt a *constraint-and-generate* strategy [27] for the construction of the CLP(FD) programs, while in ASP we exploit the natural *generate-and-test* approach [2]. Whenever possible, we make use of encodings of 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, recently published for both CLP(FD) systems [13, 34] and ASP solvers [1, 24, 21, 18].

2 Brief Overview of the Computational Models

2.1 Constraint Logic Programming over Finite Domains

Constraint logic programming (briefly, CLP) [19] is a programming paradigm that is particularly well suited for encoding combinatorial optimization problems. CLP naturally merges two declarative paradigms: constraint solving and logic programming.

Let Σ be a logic language signature $\Sigma = \langle \mathcal{F}, \mathcal{V}, \Pi \cup \Pi_C \rangle$, where

- $\mathcal{F}$ is a finite set of function and constant symbols
- $\mathcal{V}$ is a denumerable collection of variables
- $\Pi \cup \Pi_C$ is a finite set of predicate symbols, where Π and Π_C are disjoint sets.

A *constraint* is a first-order formula over $\langle \mathcal{F}, \mathcal{V}, \Pi_C \rangle$. Typically, constraints are conjunctions of literals, e.g., $0 < X, X < 3, X + Y \neq 4$. Following the traditional logic programming notation, a comma indicates a conjunction, capital letters denote variables, and the symbol $:-$ denotes the implication $\leftarrow$. CLP lets a programmer use different classes of constraints and domains to encode problems. For combinatorial problems, it is common to use *finite domain constraints*, namely arithmetic constraints between arithmetic expressions, where each variable is associated to a finite domain of possible values. In this case the interpretation of variables, expressions, and constraints is over $\mathbb{Z}$.

A CLP program over Σ is a finite set of rules of the form

$$p(s_1, \ldots, s_n) :- C, q_1(t_1^1, \ldots, t_{n_1}^1), \ldots, q_m(t_1^m, \ldots, t_{n_m}^m)$$

where C is a constraint, s_i and t_j^i are $(\mathcal{F}, \mathcal{V})$-terms, and $p, q_1, \ldots, q_m$ are predicate symbols of Π. Observe that a CLP program without constraints is in fact a Prolog program.

In contrast to classic generate-and-test approach of logic programming, CLP usually uses a *constrain-and-generate* technique in which an initial deterministic phase imposes a number of constraints, then a non-deterministic phase generates/explores the solution space. In the `constraint` phase, in particular, a finite domain of values is assigned to each of the variables. For instance, the constraint `domain([A,B,C],1,5)` assigns the set of admissible values {`1,2,3,4,5`} to the variables `A`, `B`, and `C`. The built-in predicate `labeling` implements the solution search process. Each time a variable is assigned a value, a deterministic propagation stage is executed, pruning the set of values to be attempted for the other variables. Various options (affecting, for instance, the variable selection criteria, the ordering of the attempted values, etc.) can be used to guide the search. The main structure of a program using this programming style is the following:

```
solve_problem(X1,...,Xn) :-
     constraint(X1,...,Xn),
     labeling([options], X1,...,Xn).
```

A CLP(FD) system executes program according to *goals* provided by the user, where a goal is a conjunction of literals. Given a program P and a goal $a_1, \ldots, a_k$, the program will determine the instantiations σ of the variables in the goal such that $\forall(a_1, \ldots, a_k)\sigma$ is a logical consequence of P.

2.2 Answer Set Programming

Answer Set Programming is a logic programming paradigm that has its foundations in traditional logic programming under answer set semantics [15].

Let us consider a logic language signature $\Sigma = \langle \mathcal{F}, \mathcal{V}, \Pi \rangle$, where

- $\mathcal{F}$ is a finite set of constant symbols
- $\mathcal{V}$ is a denumerable collection of variables
- Π is a finite set of predicate symbols

The set of *terms* is $\mathcal{F} \cup \mathcal{V}$, and an atom is a formula of the type $p(t_1, \ldots, t_k)$, where $p \in \Pi$ and $t_1, \ldots, t_k$ are terms.

An ASP program is a finite collection of rules, where rules are of the form

$$h :- a_1, \ldots, a_m, not\ b_1, \ldots, not\ b_n$$

where $h, a_1, \ldots, a_m, b_1, \ldots, b_n$ are atoms. Each program is viewed as a syntactic sugar representing the set of *ground instances* of its rules, where the ground instances of a rule are obtained by consistently replacing the variables in the rule with elements of $\mathcal{F}$. A special type of rules are the so called *constraints*—rules with an empty head; in this case, the head of the rule is implicitly assumed to be *false*.

Given a ground program P, the answer set semantics characterizes the semantics of P in terms of a collection of minimal models (called *answer sets*). A set of ground atoms M is an answer set if M is the unique minimal model of the program P^M, obtained as

$$P^M = \left\{ h :- a_1, \ldots, a_k \;\middle|\; \begin{array}{l} h :- a_1, \ldots, a_k, not\ b_1, \ldots, not\ b_h \in P, \\ \{b_1, \ldots, b_h\} \cap M = \emptyset \end{array} \right\}$$

In ASP, each problem is modeled as a collection of rules, in such a way that the solutions to the problem correspond one-to-one to the answer sets of the program. An *ASP Solver* is a program that computes all the answer sets of a given ASP program.

Systems like SMODELS have extended the syntax of ASP to allow more expressive constructs, such as:

- *domain declarations:* these are atoms of the form

  $$p(v_1..v_2)$$

  where $v_1 \leq v_2$ are integers; these are short forms for the collection of facts:

  $$p(v_1). \quad p(v_1 + 1). \quad \ldots \quad p(v_2).$$

- *cardinality constraint atoms:* these are atoms of the form

  $$L\{p_1, \ldots, p_k, not\ q_1, \ldots, not\ q_h\}U$$

  where $p_1, \ldots, p_k, q_1, \ldots, q_h$ are atoms and $L \leq U$ are integers; the atom is satisfied by a set of atoms M if

  $$L \leq |(\{p_1, \ldots, p_k\} \cap M) \cup (\{q_1, \ldots, q_h\} \setminus M)| \leq U$$

- *weight constraint atoms:* these are similar to cardinality atoms, but where explicit weights are attached to the elements, i.e.,

$$L[p_1 = w_1, \ldots, p_k = w_k, not\ q_1 = v_1, \ldots, not\ q_h = w_h]U$$

The atom is satisfied by M if

$$L \leq \sum_{p_i \in M} w_i + \sum_{q_i \notin M} v_i \leq U$$

Both constructs can also include *conditional atoms* of the form $p(X) : q(X)$ —these play the same role as intensional set constructors. For example, $L\{p(X) : q(X)\}U$ is treated as $L\{p(a_1), \ldots, p(a_k)\}U$ if $a_1, \ldots, a_k$ are the values for which q is true.

3 The Experimental Framework

The experimental studies reported in this paper have been mainly conducted using two CLP(FD) implementations and two ASP solvers. The CLP(FD) programs have been designed for execution by SICStus Prolog 3.11.2 (using the library `clpfd`) and GNU-Prolog 1.2.16—though the code is general enough to be used on other platforms, such as B-Prolog and ECLiPSe, with minimal syntactic adjustments [37]. 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 [36]. The CMODELS system makes use of a SAT solver to compute answer sets—in our experiments we used both the default SAT solver, mChaff [28], and Simo [17]. Some experiments have been performed using the SAT solvers zChaff and RelSat. We do not report here results concerning them—the interested reader is referred to [9] for such results.

SAT-based ASP solvers, such as CMODELS, take advantage of the *tightness* of the ASP programs [20]. In presence of non-tight programs, CMODELS is forced to repeatedly call the SAT solver in order to reach a solution. This is done to discard those models of the program that are not answer sets, trying to avoid the introduction of a potentially exponential number of *loop formulae* [23].

We focused on well-known computationally-hard problems. Among them: Graph k-coloring (Section 4), Hamiltonian circuit (Section 5), Schur numbers (Section 6), protein structure prediction in a 2D lattice [3] (Section 7), planning in a block world (Section 8), generalized Knapsack (Section 9), and code design (Section 10). While some of the programs have been drawn from the best proposals appeared in the literature, others are novel solutions, developed by the authors for this project—e.g., this is the case for the ASP implementation of the protein structure prediction problem and the planning implementation in CLP(FD).

3.1 Basic Encoding Schema

In most of the problems considered, we look for (the existence of) a function f, from a set of N elements to a set of K elements, fulfilling a collection of constraints that

characterize the solutions of the problem. To fix the ideas, let us assume that the domain of f is the set $\{1, \ldots, N\}$ while the range is the set $\{1, \ldots, K\}$.

In CLP(FD), problems of this kind are typically encoded by introducing a list `Vars` of N variables. The domain of each variable is expressed by a built-in predicate (e.g., `domain(Vars,1,K)` in SICStus). Further constraints are imposed on `Vars` depending on the specific problem at hand. The resolution of such constraint-satisfaction problem is then delegated to a *labeling* procedure, which explores the possible value-assignments for `Vars`, through some form of search space exploration (e.g., chronological backtracking).

Alternative search strategies can be typically selected by the user, via appropriate declarations. In most of the experiments we conducted, we made use of the `first-fail` strategy, i.e., the strategy that selects first the variables with the smallest domain—as it often leads to search trees with smaller degrees in the higher levels. In the Schur problem, discussed in Section 6, we report also results obtained with the `leftmost` strategy—i.e., select the leftmost uninstantiated variable from the list of problem variables. Each solution corresponds to an admissible function f. More specifically, for any $i = 1, \ldots, n$, $f(i) = j$ holds if and only if the j is assigned to the i^{th} variable in `Vars`.

The encoding style exploited in ASP is quite different. The function f is represented by a (binary) predicate `pred_f` that has to be defined so that `pred_f(i,j)` holds if and only if $f(i) = j$. Thus, a general form of this ASP-encoding is:

```
domain(1..n).
range(1..k).
1 { pred_f(X,Y) : range(Y) } 1 :- domain(X).
```

Other rules and constraints must be added to properly define `pred_f`.

We will stay as close as possible to this high-level encoding schema in all the problems we present in this paper.

In the remaining sections, we describe the solutions of 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 separately report the time needed by *lparse* to ground the programs. We used the `runtime` option to measure the time in CLP(FD), that does not account for the time spent in garbage collection and 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. The code of the various benchmarks and the complete results are reported in [9].

4 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 *"Graph Coloring and its Generalizations"* [39] repository, 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) Encoding

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. The following is a possible constrain-and-generate CLP(FD)-encoding of k-coloring:

```
 (1) coloring(K, Vars) :-
 (2)     graph(Nodes, Edges),length(Nodes,N),
 (3)     length(Vars,N),
 (4)     domain(Vars, 1, K),
 (5)     constraints(Edges, Nodes, Vars),
 (6)     labeling([ff], Vars).
 (7) constraints([],_,_).
 (8) constraints([[A,B]|R], Nodes, Vars) :-
 (9)     nth(IdfA,Nodes,A),nth(IdfA,Vars,ColA),
(10)     nth(IdfB,Nodes,B),nth(IdfB,Vars,ColB),
(11)     ColA #\= ColB,
(12)     constraints(R, Nodes, Vars).
```

In this program, `Vars` is a list of variables `[C1,...,CN]` where, for each node `i` we introduce a color variable `Coli` in the range `1...K`. Lines (3)–(5) implement the encoding schema discussed in Section 3.1. The predicate `constraints` imposes disequality constraints between variables related to adjacent nodes (line (11)). The traditional list predicate `nth` finds the positions of nodes `A` and `B` in the input list of `Nodes` and then retrieves the corresponding variables in the list `Vars` (lines (9)–(10)). We used the `ff` option of `labeling`, as it offered the best results for this problem.

ASP Encoding

In the ASP encoding of the k-coloring problem, we adopt a different representation of the graphs. Nodes are represented by facts `node(V)`, for each node `V`—as this allows a compact declaration of the nodes using the interval notation (e.g., `node(1..138)`). Edges are also encoded as 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)     1 {color(X,C):col(C)} 1 :- node(X).
(3)     :- edge(X,Y), col(C), color(X,C), color(Y,C).
```

Rule (1) states that there are `k` colors (`k` is a constant to be initialized in the grounding stage). The ASP rule (2) states that each node has exactly one color. These two lines contain the encoding schema discussed in Section 3.1. The constraint in line (3) asserts that two adjacent nodes cannot have the same color. Note that, by using domain restricted variables (ranging over all nodes), a single ASP-constraint suffices to state this property for all edges. 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 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 [39]. Such instances belong to various classes of graphs 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) spent by the various systems we used (the first column of each block indicates whether a solution exists for the problem instance). For the CLP(FD) solvers, we report only the results obtained with SICStus Prolog, because GNU-Prolog was unable to solve most of the instances (except for the smallest ones) within a reasonable amount of time. In our result tables, we will use "–" to denote the lack of answer within a given time period, while "?" indicates that none of the solvers gave an answer. As we can observe from the table, CMODELS outperforms both SMODELS and CLP(FD) in most of the tests (with very few exceptions, e.g., `mulsol`). In particular, CMODELS is capable of completing some benchmarks that the other solvers fail to complete. In most cases, the time for the program grounding is negligible w.r.t. the time for the computation of the answer sets.

A particular class of graph coloring problems listed in [39] 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 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.

5 Hamiltonian Circuit

The Hamiltonian circuit problem deals with establishing whether a directed graph admits a cycle which visits once each node in the graph. 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) Encoding

A possible CLP(FD) encoding is the following:

Instance		3-colorability						4-colorability						5-colorability					
Graph	V×E		*lparse*	SMODELS	CMODELS		CLP(FD)		*lparse*	SMODELS	CMODELS		CLP(FD)		*lparse*	SMODELS	CMODELS		CLP(FD)
					mChaff	Simo	SICStus				mChaff	Simo	SICStus				mChaff	Simo	SICStus
1-FullIns_5	282×3247	N	0.11	1.06	0.15	0.12	0.10	N	0.13	–	0.23	0.22	2.90	N	0.16	–	107.78	48.93	–
4-FullIns_4	690×6650	N	0.24	0.94	0.29	0.25	0.46	N	0.29	2.20	0.35	0.32	1.98	N	0.34	10.02	0.42	0.39	–
5-FullIns_4	1085×11395	N	0.42	1.72	0.47	0.42	1.26	N	0.51	4.67	0.57	0.53	3.58	N	0.60	23.79	0.70	0.66	–
3-FullIns_5	2030×33751	N	1.18	5.92	1.23	1.18	7.24	N	1.42	21.31	1.51	1.47	13.69	N	1.67	–	1.96	2.04	–
4-FullIns_5	4146×77305	N	2.76	15.11	2.69	2.69	33.44	N	3.35	69.30	3.37	3.38	42.53	N	3.95	414.93	4.19	4.13	–
3-Insertions_3	56×110	N	<0.01	4.28	4.16	7.59	1281.18	Y	0.01	0.03	0.04	0.01	<0.01	Y	0.01	0.04	0.04	0.01	<0.01
4-Insertions_3	79×156	N	0.01	328.25	1772.14	1481.27	–	Y	0.01	0.05	0.04	0.01	<0.01	Y	0.01	0.06	0.05	0.02	<0.01
2-Insertions_4	149×541	N	0.02	1.20	0.15	1.08	2.04	?	0.02	–	–	–	–	Y	0.03	0.25	0.07	0.04	0.01
4-Insertions_4	475×1795	N	0.08	–	1443.33	1468.91	–	?	0.09	–	–	–	–	Y	0.11	3.402	0.32	0.35	–
2-Insertions_5	597×3936	N	0.15	45.08	0.50	2.62	6.97	?	0.18	–	–	–	–	?	0.22	–	–	–	–
DSJR500.1	500×3555	N	0.13	0.53	0.18	0.15	0.18	N	0.16	2.78	0.21	0.19	0.18	N	0.19	–	0.26	0.23	0.19
DSJC500.1	500×12458	N	0.42	2.19	0.45	0.42	0.64	N	0.51	12.30	0.57	0.57	0.76	N	0.61	–	6.21	120.47	46.55
DSJR500.5	500×58862	N	1.87	25.76	1.81	1.78	2.97	N	2.28	175.63	2.26	2.19	2.98	N	2.69	971.46	2.71	2.68	3.09
DSJC500.5	500×62624	N	2.02	28.29	1.92	1.92	3.15	N	2.45	376.35	2.36	2.37	3.19	N	2.80	–	2.84	2.99	3.47
DSJR500.1c	500×121275	N	3.81	84.19	3.66	3.68	6.07	N	4.71	1083.17	4.54	4.59	6.18	N	5.51	–	5.50	5.53	6.19
DSJC500.9	500×224874	N	3.57	74.44	3.39	3.42	5.67	N	4.35	543.02	4.29	4.32	5.67	N	5.12	–	5.09	5.05	5.77
DSJC1000.1	1000×49629	N	1.61	12.99	1.61	1.62	5.01	N	1.97	241.43	2.02	2.23	5.06	N	2.32	–	3.61	98.85	–
flat300_20_0	300×21375	N	0.67	6.39	0.68	0.66	0.63	N	0.82	86.91	0.84	0.82	0.64	N	0.97	1555.37	1.08	1.03	0.69
flat300_26_0	300×21633	N	0.67	6.45	0.70	0.66	0.65	N	0.83	131.91	0.87	0.84	0.67	N	0.99	–	1.13	1.15	0.69
flat300_28_0	300×21695	N	0.67	6.51	0.70	0.68	0.65	N	0.83	34.76	0.86	0.83	0.69	N	0.98	322.99	1.02	1.01	0.67
fpsol2.i.1	496×11654	N	0.39	2.75	0.41	0.37	0.77	N	0.48	24.98	0.52	0.48	0.77	N	0.57	205.12	0.61	0.57	0.84
fpsol2.i.2	451×8691	N	0.28	1.92	0.33	0.29	0.53	N	0.35	16.66	0.40	0.37	0.54	N	0.41	279.96	0.52	0.46	0.55
fpsol2.i.3	425×8688	N	0.28	1.91	0.32	0.29	0.5	N	0.34	16.63	0.40	0.36	0.51	N	0.40	277.91	0.49	0.46	0.51
gen200_p0.9_44	200×17910	N	0.56	5.53	0.57	0.55	0.36	N	0.68	30.87	0.70	0.68	0.36	N	0.79	306.81	0.84	0.82	0.38
gen200_p0.9_55	200×17910	N	0.56	5.54	0.57	0.54	0.36	N	0.68	39.56	0.71	0.68	0.36	N	0.79	287.14	0.85	0.81	0.38
gen400_p0.9_55	400×71820	N	2.27	38.91	2.19	2.18	2.88	N	2.81	656.07	2.68	2.69	2.89	N	3.23	–	3.24	3.22	2.93
gen400_p0.9_65	400×71820	N	2.27	39.02	2.16	2.15	2.88	N	2.76	275.33	2.67	2.66	2.87	N	3.24	1563.52	3.22	3.23	2.92
gen400_p0.9_75	400×71820	N	2.26	38.87	2.17	2.15	2.88	N	2.76	270.12	2.70	2.71	2.89	N	3.25	1608.19	3.22	3.19	2.94
inithx.i.1	864×18707	N	0.62	4.92	0.65	0.65	2.28	N	0.76	57.15	0.81	0.78	2.29	N	0.89	415.41	1.00	0.93	2.32
inithx.i.2	645×13979	N	0.47	3.50	0.50	0.46	1.28	N	0.56	34.19	0.63	0.59	1.28	N	0.67	268.87	0.83	0.73	1.31
inithx.i.3	621×13969	N	0.46	3.50	0.50	0.52	1.22	N	0.55	34.14	0.64	0.58	1.24	N	0.66	268.36	0.80	0.73	1.26
le450_5a	450×5714	N	0.20	0.85	0.24	0.21	0.26	N	0.25	9.06	0.29	0.27	0.28	Y	0.28	190.38	12.30	1.99	5.29
le450_5b	450×5734	N	0.20	0.85	0.24	0.20	0.29	N	0.24	7.77	0.29	0.27	0.3	Y	0.28	–	0.98	12.47	0.48
le450_5c	450×9803	N	0.32	1.64	0.37	0.33	0.44	N	0.39	9.98	0.46	0.45	0.44	Y	0.46	217.77	0.70	0.58	0.03
le450_5d	450×9757	N	0.32	1.64	0.36	0.33	0.44	N	0.39	10.29	0.44	0.42	0.47	Y	0.46	530.63	0.60	0.51	0.08
mulsol.i.1	197×3925	N	0.13	0.58	0.17	0.12	0.10	N	0.16	2.80	0.20	0.16	0.10	N	0.19	19.07	0.24	0.19	0.11
mulsol.i.2	188×3885	N	0.12	0.57	0.17	0.13	0.10	N	0.15	2.83	0.20	0.16	0.09	N	0.18	31.25	0.24	0.19	0.11
mulsol.i.3	184×3916	N	0.13	0.58	0.16	0.12	0.09	N	0.15	2.86	0.20	0.16	0.10	N	0.18	31.93	0.25	0.20	0.10
mulsol.i.4	185×3946	N	0.12	0.58	0.17	0.13	0.10	N	0.15	2.92	0.20	0.17	0.10	N	0.19	32.83	0.25	0.20	0.11
mulsol.i.5	186×3973	N	0.13	0.59	0.17	0.13	0.10	N	0.15	2.93	0.20	0.16	0.09	N	0.18	33.02	0.26	0.20	0.11
wap05a	905×43081	N	1.39	11.39	1.38	1.36	2.96	N	1.69	62.81	1.73	1.70	2.96	N	2.00	949.66	2.07	2.05	2.96
wap06a	947×43571	N	1.38	11.63	1.42	1.40	3.25	N	1.70	62.70	1.75	1.70	3.24	N	2.01	1326.84	2.13	2.10	3.26
wap07a	1809×103368	N	3.39	31.98	3.28	3.31	15.14	N	4.13	191.06	4.12	4.09	15.14	N	4.87	–	4.99	5.05	15.19
wap08a	1870×104176	N	3.48	32.07	3.31	3.32	16.17	N	4.25	192.54	4.15	4.19	16.22	N	5.02	–	5.08	5.00	16.18

Table 1: Graph k-coloring (30 minutes time limit)

Instance		Solvability for $M = N - 1$					Solvability for $M = N$					Solvability for $M = N + 1$				
N	$V \times E$		SMODELS	CMODELS mChaff	SICStus CLP(FD)	SICStus ugraphs		SMODELS	CMODELS mChaff	SICStus CLP(FD)	SICStus ugraphs		SMODELS	CMODELS mChaff	SICStus CLP(FD)	SICStus 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	–
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 (10 minutes time limit)

```
 (1) hc(Path) :-
 (2)     graph(Nodes, Edges),length(Nodes, N),
 (3)     length(Path, N),
 (4)     set_domains(Path, 1, Edges),
 (5)     circuit(Path),
 (6)     labeling([ff], Path).
 (7) set_domains([], _, _ ).
 (8) set_domains([X_I | Path], I, Edges) :-
 (9)     findall(Z, member([I,Z], Edges), [S|Ucc]),
(10)     convert([S|Ucc],Domain),
(11)     X_I in Domain,
(12)     I1 is I+1, set_domains(Path, I1, Edges).
(13) convert([A],{A}) .
(14) convert([A|R],{A} \/ S) :- convert(R,S).
```

In this case, we generate a list `Path` of `N` variables (line (3)). The expected value of variable X_I is the successor of the node I in the Hamilton circuit. Therefore, its domain is the set of all nodes Z that can be reached from node I with an edge $\langle X_I, Z\rangle$. The list of these nodes is computed in line (9). It is then converted in a suitable representation for `clpfd` (using the predicate `convert`, defined in lines (13) and (14)). The domain for X_I is established in line (11). Just a remark on the requirement of non-empty list of successors in line (9): if a node has no successors there cannot be any Hamiltonian circuits, and thus `hc` will fail. On the other hand, the `in` built-in of line (11) requires a non-empty set on the right-hand side to be well-typed.

We use the built-in predicate `circuit`, provided by `clpfd` in SICStus. In the literal `circuit(Path)`, the `Path` is the above mentioned list of finite domain variables. The goal `circuit`$([X_1, \ldots, X_n])$ constrains the variables so that the set of edges $\langle 1, X_1\rangle$, $\langle 2, X_2\rangle, \ldots, \langle n, X_n\rangle$ represents an Hamiltonian circuit.

ASP Encoding

The ASP program for Hamiltonian circuit has been drawn from the ASP literature [26]:

```
(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).
```

Instance	nodes×edges	Hamiltonian?					
			lparse	SMODELS	CMODELS		SICStus
					mChaff	Simo	
hc1	200×1250	Y	0.46	3.01	37.60	14.30	0.33
hc2	200×1250	Y	0.44	3.03	1390.15	3.48	0.35
hc3	200×1250	Y	0.45	3.06	20.08	6.52	0.31
hc4	200×1250	Y	0.45	3.02	92.99	6.54	0.33
hc5	200×1250	N	0.44	1.45	0.22	0.21	0.24
hc6	200×1250	N	0.45	1.46	0.21	0.21	0.10
hc7	200×1250	N	0.45	1.46	0.21	0.21	0.24
hc8	200×1250	N	0.45	1.46	0.21	0.21	0.23
np10c	10×90	Y	<0.01	0.01	0.04	0.01	<0.01
np20c	20×380	Y	0.01	0.07	0.81	0.05	0.01
np30c	30×870	Y	0.03	0.26	0.26	0.20	0.01
np40c	40×1560	Y	0.07	0.84	4.35	0.68	0.02
np50c	50×2450	Y	0.11	2.69	117.77	1.55	0.03
np60c	60×3540	Y	0.17	7.34	24.72	5.54	0.04
np70c	70×4830	Y	0.25	15.58	9.46	7.46	0.07
np80c	80×6320	Y	0.34	27.88	12.53	12.60	0.10
np90c	90×8010	Y	0.45	46.07	127.31	42.32	0.14
2xp30	60×316	N	0.04	0.14	0.02	0.02	0.01
2xp30.1	60×318	Y	0.04	0.19	4.59	377.09	0.03
2xp30.2	60×318	Y	0.04	7931.75	2.68	3.90	5.42
2xp30.3	60×318	Y	0.04	7910.75	2.69	3.87	5.39
2xp30.4	60×318	N	0.04	–	5692.11	–	–
4xp20	80×392	N	0.07	0.24	0.04	0.04	0.04
4xp20.1	80×395	N	0.07	–	1.48	36.36	0.03
4xp20.2	80×396	Y	0.07	0.37	3.32	3.42	0.05
4xp20.3	80×396	N	0.07	0.24	2.62	51.45	–

Table 3: Hamiltonian circuit (180 minutes time limit)

```
(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): 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 exploit the built-ins `circuit` and `findall`).

Results

Most of the problem instances have been taken from the benchmarks used to compare ASP solvers [24]. 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 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 [36]. Table 3 reports the experimental results.

Note that the ASP encoding we used originates non-tight programs. Hence, as expected, the performances of SMODELS and CMODELS are closer than in the other problems (which, actually, involve tight programs) SMODELS performs well in various

instances, outperforming CMODELS in the relatively smaller instances of the hc graphs and in most of the np graphs. SICStus outperforms the ASP solvers in most of the cases.

6 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 $160 \leq S(5) \leq 315$ [35]. We focus on the decision problem: is $S(P) \geq N$? Namely, we look for a function $B : [1..N] \longrightarrow [1..P]$ such that: $(\forall I \in [1..N])(\forall J \in [I, \ldots, N])(B(I) = B(J) \rightarrow B(I + J) \neq B(I))$.

CLP(FD) Encoding

The encoding of this problem in CLP(FD) makes use of a list of constrained variables `List = [B1,...,BN]`, each having the domain `1,...,P`.

```
 (1) schur(N,P) :-
 (2)      length(List,N),
 (3)      domain(List,1,P),
 (4)      constraints(List,N),
 (5)      labeling([ff],List). %%% labeling([leftmost],List).
 (6) constraints(List, N) :-
 (7)      List=[1,2|_],
 (8)      recursion(List,1,1,N).
 (9) recursion(_,I,_,N):-  I>N, !.
(10) recursion(List,I,J,N):-
(11)      I+J>N, !,
(12)      I1 is I+1, recursion(List,I1,1,N).
(13) recursion(List,I,J,N):-
(14)      I>J, !,
(15)      J1 is J+1, recursion(List,I,J1,N).
(16) recursion(List,I,J,N):-
(17)      K is I+J, J1 is J+1,
(18)      nth(I,List,BI), nth(J,List,BJ), nth(K,List,BK),
(19)      (BI #= BJ) #=> (BK #\= BI),
(20)      recursion(List,I,J1,N).
```

Each variable `Bi` in `List` can assume values in `1..P` (line (3)). 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. Line (7) sets `B1 = 1` and `B2 = 2` to remove some simple symmetries; this is required to allow for a fair comparison w.r.t. the ASP solution, that imposes an analogous constraint

Instance $\langle P, N\rangle$	is $Schur(P) \geq N$?										
		lparse	SMODELS	CMODELS		SICStus				GNU-Prolog	
				mChaff	Simo	first-fail	leftmost	first-fail + (7)	leftmost + (7)	first-fail + (7)	leftmost + (7)
$\langle 4, 43\rangle$	Y	0.06	0.27	0.25	1.59	2.49	0.04	2.78	0.03	0.31	<0.01
$\langle 4, 44\rangle$	Y	0.06	0.29	3.37	3.50	4.14	20.84	4.44	20.72	0.56	0.97
$\langle 4, 45\rangle$	N	0.06	510.01	892.54	299.33	–	–	–	1204.86	–	53.04
$\langle 4, 46\rangle$	N	0.07	561.80	813.73	196.47	–	–	–	1340.64	–	54.58
$\langle 4, 47\rangle$	N	0.07	767.80	791.37	295.70	–	–	–	1473.02	–	56.16
$\langle 4, 48\rangle$	N	0.07	978.84	805.69	312.28	–	–	–	1565.28	–	57.11
$\langle 4, 49\rangle$	N	0.07	1258.57	679.20	216.00	–	–	–	1698.08	–	58.42
$\langle 5, 109\rangle$	Y	0.47	–	14.05	0.90	0.46	0.18	0.43	0.13	0.05	0.05
$\langle 5, 110\rangle$	Y	0.48	–	33.29	0.84	54.75	0.16	55.08	0.14	5.78	0.05
$\langle 5, 111\rangle$	Y	0.48	–	0.53	14.69	57.14	0.14	57.19	0.16	5.80	0.05
$\langle 5, 112\rangle$	Y	0.50	–	0.55	0.73	58.46	0.18	58.56	0.16	5.73	0.05
$\langle 5, 113\rangle$	Y	0.51	–	0.55	0.85	207.80	0.19	207.98	0.16	3.79	0.05
$\langle 5, 114\rangle$	Y	0.52	–	11.75	0.77	1037.88	0.16	1038.86	0.16	3.93	0.05
$\langle 5, 115\rangle$	Y	0.52	–	82.48	1.22	1074.24	8.53	1076.42	8.63	4.27	0.51
$\langle 5, 116\rangle$	Y	0.53	–	60.47	1.06	1110.47	8.88	1116.71	8.92	4.44	0.51
$\langle 5, 117\rangle$	Y	0.54	–	762.91	0.81	1149.36	9,62	1156.06	9.74	2.62	0.56
$\langle 5, 118\rangle$	Y	0.55	–	21.84	5.17	1192.68	10.16	1197.82	10.19	4.68	0.56
$\langle 5, 119\rangle$	Y	0.57	–	–	8.44	1233.69	66.75	1239.31	66.95	5.19	3.59

Table 4: Schur numbers (30 minutes time limit)

(see lines (4) and (5) in the ASP-encoding of this problem).

ASP Encoding

The above mentioned function B is implemented by a predicate `inpart(X,P)`, which represents the fact that the number `X` is assigned to the block `P` of the partition:

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

The declarative formalization of the problem is composed of rules (1) to (4). Rule (3) states that `inpart` is a function from numbers to blocks of the partition. The ASP-constraint (4) states that, for any `X` and `Y`, the three numbers `X`, `Y`, and `X+Y` cannot belong to the same block. It is also customary to add the constraints (5) and (6), that remove symmetries, by selecting, for each number, the free block with the lowest index.

Results

Table 4 reports the execution time results for different instances of the problem. The second column indicates whether the answer to the test is positive or negative. For CLP(FD), we report results using both `leftmost` and `first-fail` labeling strategies; we also provide results with and without the constraint of line (7). The leftmost strategy, along with constraint (7), allows the CLP(FD) solvers to handle all instances of the problem (while the first-fail strategy is unable to handle the cases with a negative

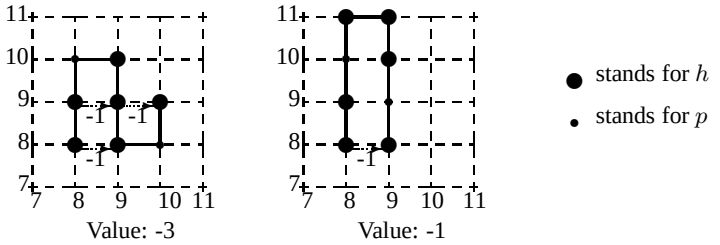

Figure 1: Two foldings for $S = hhphhhph$ ($n = 8$). The leftmost one is minimal.

answer)—it appears that in this problem the solution lies towards the left part of the tree. Observe that GNU-Prolog significantly outperforms all other solvers.

7 Protein Structure Prediction

The problem we study is an abstraction of the protein structure prediction problem. In the *2D HP-protein structure prediction problem* [3], given a protein sequence, we separate the amino acids in two classes, h (hydrophobic) and p (polar, or hydrophilic), and we try to determine a folding of the sequence in the 2D square lattice space such that most HH pairs are neighboring. More precisely, given a sequence $S = s_1 \cdots s_n$, with $s_i \in \{h, p\}$, the *2D, HP-protein structure prediction problem* [3] is the problem of finding a mapping (*folding*) $\omega : \{1, \ldots, n\} \longrightarrow \mathbb{N}^2$ such that

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

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 further reduce the solution space, we imposed 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 (amino acids) 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) Encoding

A complete CLP(FD) encoding of this problem (following the general design presented in [3]) is presented below. Lines (1)–(3) drive the basic constraint-and-generate

scheme. The predicate `constrain` introduces the list of unknown variables (lines (5)–(6)), set their domains (lines (7)–(8)) and call the other constraints predicates. The predicate `starting_points` (line (13)) sets the two initial points to be $\langle N, N\rangle$ and $\langle N, N+1\rangle$. The predicate `avoid_self_loops` (line (14)) forces all points $\langle X_i, Y_i\rangle$ and $\langle X_j, Y_j\rangle$, with $i \neq j$ to be different. This is stated, for each pair, by a unique integer constraint: $kX_i + Y_i \neq kX_j + Y_j$, where k can be any number sufficiently larger than N (we used 100 in line (19)). The predicate `next_constraints` states that $\texttt{abs}(X_i - X_{i+1}) + \texttt{abs}(Y_i - Y_{i+1}) = 1$ for all $i = 1, .., N-1$ (lines (26)–(28)). The predicate `energy_constraints` adds the energy contribution of every pair $\langle X_i, Y_i\rangle$ and $\langle X_j, Y_j\rangle$, with $j > i+1$. This contribution is always 0 but in the case $s_i = s_j = h$ and $\texttt{abs}(X_i - X_j) + \texttt{abs}(Y_i - Y_j) = 1$ (lines (41)–(46)).

```
 (1) pf(Primary, Tertiary) :-
 (2)    constrain(Primary,Tertiary,Energy),
 (3)    labeling([ff,minimize(Energy)],Tertiary).
 (4) constrain(Primary,Tertiary,Energy) :-
 (5)    length(Primary,N),
 (6)    M is 2*N,length(Tertiary,M),
 (7)    Min is N-integer(sqrt(N)), Max is N+integer(sqrt(N)),
 (8)    domain(Tertiary,Min,Max),
 (9)    starting_point(Tertiary,N),
(10)    avoid_self_loops(Tertiary),
(11)    next_constraints(Tertiary),
(12)    energy_constraint(Primary,Tertiary,Energy).
(13) starting_point([N,N,N,N1|_],N) :- N1 is N+1.
(14) avoid_self_loops(Tertiary):-
(15)    positions_to_integers(Tertiary, ListaInteri),
(16)    all_different(ListaInteri).
(17) positions_to_integers([],[]).
(18) positions_to_integers([X,Y|R], [I|S]):-
(19)    I #= X*100+Y,
(20)    positions_to_integers(R,S).
(21) next_constraints([_,_]).
(22) next_constraints([X1,Y1,X2,Y2|C]) :-
(23)    next(X1,Y1,X2,Y2),
(24)    next_constraints([X2,Y2|C]).
(25) next(X1,Y1,X2,Y2):-
(26)    domain([Dx,Dy],0,1),
(27)    Dx #= abs(X1-X2), Dy #= abs(Y1-Y2),
(28)    Dx+Dy #= 1.
(29) energy_constraint([_],_,0).
(30) energy_constraint([A,B|Primary],[XA,YA,XB,YB|Tertiary],E) :-
(31)    energy_contrib(0,A,XA,YA,Primary,Tertiary,E1),
(32)    energy_constraint([B|Primary],[XB,YB|Tertiary],E2),
(33)    E #= E1+E2.
(34) energy_contrib(_,_,_,_,[],[],0).
(35) energy_contrib(0,A,XA,YA,[_|Primary],[_,_|Tertiary],E):-
(36)    energy_contrib(1,A,XA,YA,Primary,Tertiary,E).
(37) energy_contrib(1,A,XA,YA,[B|Primary],[XB,YB|Tertiary],E):-
(38)    energy(A,XA,YA,B,XB,YB,C), E #= E1+C,
```

```
(39)    energy_contrib(0,A,XA,YA,Primary,Tertiary,E1).
(40) energy(h,XA,YA,h,XB,YB,C) :-
(41)    DX #= abs(XA-XB), DY #= abs(YA-YB),
(42)    C in {0,-1},
(43)    1 #= DX+DY  #<=> C #= -1.
(44) energy(h,_,_,p,_,_,0).
(45) energy(p,_,_,h,_,_,0).
(46) energy(p,_,_,p,_,_,0).
```

A refinement of this encoding (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 Encoding

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 by a set of facts, describing the sequence of amino acids. For instance, the protein denoted by `hpphpphpph` ($(\texttt{hpp})^3\texttt{h}$, for short) 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), 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 amino acid occupies exactly one position. Rules (3) and (4) fix the positions of the two initial amino acids (they eliminate some symmetric solutions). The ASP-constraints (6) and (7) state that there are no self-loops and that two contiguous amino acids 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 amino acids, 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, then the assertion (10) is replaced by

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

Instance			Optimization problem			
Sequence	Length	Min	*lparse*	SMODELS	SICStus	GNU-Prolog
h^{10}	10	-4	0.08	0.74	0.13	0.03
h^{15}	15	-8	0.20	10.61	5.50	3.15
h^{20}	20	-12	0.61	1679.79	766.22	148.53
h^{25}	25	-16	1.50	–	103962.57	19266.30
$(hpp)^3h$	10	-4	0.08	0.51	0.10	<0.01
$(hpp)^5h$	16	-6	0.39	22.33	0.22	0.15
$(hpp)^7h$	22	-8	0.74	1059.86	46.87	38.18
$(hpp)^9h$	28	-10	1.86	–	14007.07	6574.70

Table 5: Protein structure prediction optimization problem (1800 minutes time limit)

Instance			Decision problem					
Sequence	Length	Min	*lparse*	SMODELS	CMODELS		SICStus	GNU-Prolog
					mChaff	Simo		
h^{10}	10	-4	0.09	0.53	1.01	0.62	<0.01	<0.01
h^{15}	15	-8	0.19	2.29	2.73	5.29	0.05	0.11
h^{20}	20	-12	0.61	28.23	52.43	1685.03	0.50	0.20
h^{25}	25	-16	1.47	2169.49	2620.94	–	1664.35	659.43
$(hpp)^3h$	10	-4	0.09	0.35	0.33	0.97	<0.01	<0.01
$(hpp)^5h$	16	-6	0.39	16.06	15.94	34.34	0.08	0.07
$(hpp)^7h$	22	-8	0.74	96.45	1609.75	84.67	5.52	8.40
$(hpp)^9h$	28	-10	1.83	2309.14	5813.23	424.56	2815.62	1951.56

Table 6: Protein structure prediction decision problem (420 minutes time limit)

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) justifies viewing it as an optimization problem. The lack of support for optimization statements in CMODELS prevents us from using this ASP solver for this problem—thus, our comparison in Table 5 focuses only on SMODELS and the CLP(FD) solvers. The table indicates, for each experiment, the length of the sequence, the minimum value of the energy, and the various execution times. The CLP(FD) solvers significantly outperform SMODELS (which is also unable to complete the largest instance of the problem within the established time limit). In turn, GNU-Prolog provides a significantly better performance.

We also performed a series of tests relative to the decision version of this problem, i.e., 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 reported in Table 6. It is interesting to observe how the performance degrades more quickly as the problem size increases for all solvers except CMODELS (with the Simo SAT solver).

8 Planning

One of the most successful applications of ASP has been in the domain of reasoning about actions and change [2]. Planning problems have been effectively encoded as ASP programs—where distinct answer sets represent different trajectories leading to the desired goal. CLP(FD) has been used less frequently for planning problems (e.g., [33]).

A planning problem is based on the notions of `State`—i.e., a representation of the relevant components of the world—and `Actions`—that affect the state of the world, and thus allow the transition from a state to another. Each state is described by the truth value assigned to a given collection of atomic formulae (called *fluents*). Planning domains can be effectively described using high-level *action description languages*, e.g., the languages $\mathcal{A}$ and $\mathcal{B}$ [16]. In our experiments, we use a slight variation of $\mathcal{B}$, based on a syntax similar to the one used in [32]. An *action theory* is a specification of a planning problem using the action language.

Let us review the structure of the action descriptions used in our tests. An assertion of the kind `fluent(f)` declares that `f` is a fluent. Fluent literals are constructed from fluents and their negations (denoted by `mneg(f)`). Declarations of the form `action(a)` are used to describe the possible actions (in this case, `a`). The language $\mathcal{B}$ allows one to specify an action theory relating actions, states, and fluents using predicates of the following forms (where `[list-of-preconditions]` denotes a list of fluent literals):

- `executable(a, [list-of-preconditions])` asserting that the given preconditions have to be satisfied in the current state in order for the action `a` to be executable.
- `causes(a, f, [list-of-preconditions])` encodes a dynamic causal law, describing the effect (the fluent literal `f`) of the execution of action `a` in a state satisfying the given preconditions.
- `caused([list-of-preconditions], f)` describes a static causal law—i.e., the fact that the fluent literal `f` is true in a state satisfying the given preconditions.

A specific instance of a planning problem contains also a description of the initial state and of the desired goal:

- `initially(f)` asserts that the fluent literal `f` is true in the initial state. In our examples, we assume complete initial states (i.e., we have knowledge of the truth value of each fluent in the initial state).
- `goal(f)` asserts that the goal requires the fluent literal `f` to hold in the final state.

Our approach assumes that the length of the desired plan is given; we also disallow parallel actions. In the following, we describe how to solve a planning problem specified in this action language, using CLP(FD) and ASP solvers.

As an example of the planning problems we experimented with, we describe a planning problem in the block world domain with 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. An additional restriction must be met: in each

state at most three blocks can lie on the table. The following is the specification of an instance of the planning problem with $N = 5$ using the proposed action language.

```
 (1) blk(1).  blk(2).  blk(3).  blk(4).  blk(5).
 (2) fluent(on_table(X)):- blk(X).
 (3) fluent(clear(X)):- blk(X).
 (4) fluent(on(X,Y)):- blk(X), blk(Y), neq(X,Y).
 (5) fluent(space_on_table).
 (6) action(move(X,Y)):- blk(X), blk(Y), neq(X,Y).
 (7) action(to_table(X)):- blk(X).
 (8) initially(clear(5)).
 (9) initially(mneg(clear(X))) :- blk(X), X<5.
(10) initially(on_table(1)).
(11) initially(mneg(on_table(X))) :- blk(X), X>1.
(12) initially(space_on_table).
(13) initially(on(X,Y)) :- blk(X), blk(Y), Y<5, X is Y+1.
(14) initially(mneg(on(X,Y))):- blk(X), blk(Y), X<Y.
(15) initially(mneg(on(X,Y))) :- blk(X), blk(Y), Y<5,
       neq(Y,X), P is Y+1, neq(X,P).
(16) goal(on(X,Y)) :- blk(X), blk(Y), Y<4, X is Y+2.
(17) goal(on_table(1)).
(18) goal(on_table(2)).
(19) goal(space_on_table).
(20) causes(move(X,Y),clear(Z),[on(X,Z)]):- blk(X), blk(Y), blk(Z),
       action(move(X,Y)),neq(X,Z),neq(Y,Z).
(21) causes(move(X,Y),on(X,Y),[]):- blk(X), blk(Y), action(move(X,Y)).
(22) causes(move(X,Y),mneg(on(X,Z)),[on(X,Z)]):- blk(X), blk(Y), blk(Z),
       action(move(X,Y)),neq(Y,Z).
(23) causes(move(X,Y),space_on_table,[on_table(X)]):- blk(X), blk(Y),
       action(move(X,Y)).
(24) causes(to_table(X),on_table(X),[]):- blk(X), action(to_table(X)).
(25) causes(to_table(X),clear(Y),[on(X,Y)]):- blk(X), blk(Y),
       neq(X,Y), action(to_table(X)).
(26) causes(to_table(X),mneg(space_on_table),[on_table(Y),on_table(Z)]):-
       blk(X), blk(Y), blk(Z), neq(X,Y), neq(Y,Z), neq(X,Z),
       action(to_table(X)).
(27) executable(move(X,Y),[clear(X),clear(Y)]) :- blk(X), blk(Y),
       action(move(X,Y)).
(28) executable(to_table(X),[clear(X),mneg(on_table(X)),space_on_table]):-
       blk(X), action(to_table(X)).
(29) caused([on(X,Y)],mneg(clear(Y))) :- blk(X), blk(Y), neq(X,Y).
(30) caused([clear(Y)],mneg(on(X,Y))) :- blk(X), blk(Y), neq(X,Y).
(31) caused([on(X,Y)],mneg(on_table(X))) :- blk(X), blk(Y), neq(X,Y).
(32) caused([on_table(X)],mneg(on(X,Y))) :- blk(X), blk(Y), neq(X,Y).
(33) caused([on(X,Y)],mneg(on(Y,X))) :- blk(X), blk(Y), neq(X,Y).
```

In this specification, the blocks are defined in line (1). Lines (2)–(5) define the fluents of the problem. A block may lie on the table or on top of another block (fluent `on(X,Y)`). A block may be clear (if no other block is on top of it) or not. There may be space on the table for other blocks (fluent `space_on_table`). There are two possible moves (lines (6)–(7)). The initial state and the goal state are described in lines (8)–(15) and (16)–(19), respectively. The `causes` rules (lines (20)–(26)) assert the effect of moves on the fluents. Executability predicates (lines (27)–(28)) impose that one can move a block `X` on top of another block `Y` only if both are clear. On the other hand, `X` can be moved to the table only if there is free space (i.e., there are at most two blocks

lying on the table). The `caused` assertions (lines (29)–(33)) are easy to understand.

In the following we describe two ways of solving planning problems specified in the proposed action language, using a CLP(FD) solver and an ASP solver. More specifically, regarding CLP(FD), we report a program that directly processes an action theory specification. Concerning the ASP solution, we developed a translator, written in Prolog, that, given an action theory specification, produces a suitable input file for *lparse*—the translation process follows the general guidelines highlighted, for example, in [32, 14].

CLP(FD)

Our CLP(FD) encoding of a planning problem uses the following representation. A plan with N states, p different fluents, and m possible actions is represented by:

- A list `States` of N lists of p terms `fluent(fluent_name, Bool_var)`. The variable of the i^{th} term in the j^{th} list is 1 if and only if the i^{th} fluent holds in the j^{th} state. E.g., with $N = 3$ and fluents `f`, `g`, and `h`, we have:

```
States = [[fluent(f,X_f_1),fluent(g,X_g_1),fluent(h,X_h_1)],
          [fluent(f,X_f_2),fluent(g,X_g_2),fluent(h,X_h_2)],
          [fluent(f,X_f_3),fluent(g,X_g_3),fluent(h,X_h_3)]]
```

- A list `ActionsOcc` of $N-1$ lists of m terms `action(action_name,Bool_var)`. The variable of the i^{th} term of the j^{th} list is 1 if and only if the i^{th} action occurs during the transition from state i to state $i + 1$. E.g., with $N = 3$ and actions `a` and `b`, we have:

```
ActionsOcc = [[action(a,X_a_1),action(b,X_b_1)],
              [action(a,X_a_2),action(b,X_b_2)]]
```

The CLP program listed below takes an action language specification as input and searches for a plan. In particular, lines (2) and (3) collect the lists of fluents (`Lf`) and actions (`La`), the description of the initial (`Init`) and the final (`Goal`) states. Lines (4) and (5) define the lists `States` and `ActionsOcc`, as explained above. The predicates in lines (6) and (7) handle the knowledge about the initial and the goal states, respectively. Lines (8) and (9) impose the constraints on state transitions and action executability. Line (10) gathers all variables denoting action occurrences, in preparation for the labeling phase (line (11)). Note that the labeling is focused on the selection of the action to be executed at each time step. The clauses in lines (12)–(36) define the predicates used in lines (4)–(7).

```
(1) main(N,ACTIONSOCC,STATES):-
(2)   setof(F,fluent(F),Lf), setof(A,action(A),La),
(3)   setof(F,initially(F),Init), setof(F,goal(F),Goal),
(4)   make_states(N,Lf,STATES),
(5)   make_action_occurrences(N,La,ACTIONSOCC),
(6)   set_initial(Init,STATES),
(7)   set_goal(Goal,STATES),
(8)   set_transitions(ACTIONSOCC,STATES),
```

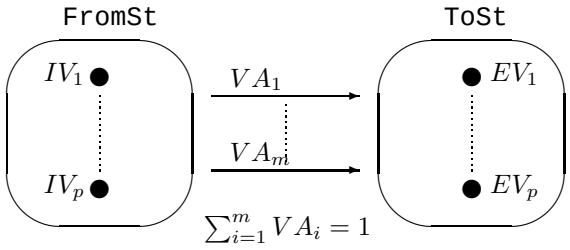

Figure 2: Action constraints (see procedures `set_transitions`—`build_sum_prod`)

```
 (9)   set_executability(ACTIONSOCC,STATES),
(10)   get_all_actions(ACTIONSOCC,AllActions),
(11)   labeling([ff],AllActions).
(12) make_states(0,_,[]).
(13)   make_states(N,List,[S|STATES]) :-
(14)   N1 is N-1, make_states(N1,List,STATES),
(15)   make_one_state(List,S).
(16) make_one_state([],[]).
(17) make_one_state([F|Fluents],[fluent(F,VarF)|VarFluents]) :-
(18)   make_one_state(Fluents,VarFluents), VarF in 0..1.
(19) make_action_occurrences(1,_,[]).
(20) make_action_occurrences(N,List,[Act|ActionsOcc]) :-
(21)   N1 is N-1, make_action_occurrences(N1,List,ActionsOcc),
(22)   make_one_action_occurrences(List,Act),
(23)   get_action_list(Act,AList), sum(AList,#=,1).
(24) make_one_action_occurrences([],[]).
(25) make_one_action_occurrences([A|Actions],[action(A,OccA)|OccActs]) :-
(26)   make_one_action_occurrences(Actions,OccActs), OccA in 0..1.
(27) set_initial(List,[InitialState|_]) :-
(28)   set_state(List,InitialState).
(29) set_goal(List,States) :-
(30)   last(States,FinalState),
(31)   set_state(List,FinalState).
(32) set_state([],_).
(33) set_state([Fluent|Rest],State) :-
(34)    (Fluent=mneg(F),!,member(fluent(F,0),State);
(35)     member(fluent(Fluent,1),State)),
(36)    set_state(Rest,State).
(37)                                  %...continued
```

The core of the constraint phase is carried out by the predicates `set_transitions` and `set_executability` (lines (8)–(9)). In particular, lines (37)–(44) recursively define the predicate `set_transitions`, which ultimately relies on `set_one_fluent` to constrain all boolean variables related to each possible fluent in each possible state transition (i.e., action execution).

Let us consider a state transition from state `FromSt` to state `ToSt`, as depicted in Figure 2, where we assume that there are p fluents $1, \ldots, p$ involved. As mentioned, two boolean variables IV_i and EV_i (for $i = 1, \ldots, p$) denote whether the fluent i holds in `FromSt` and in `ToSt`, respectively. Let the variable VA_j, ($j = 1, \ldots, m$) denote whether the action j occurs in such a state transition. In this situation, let us consider a generic call (line (43)) of the predicate `set_one_fluent` (defined in line (45)). For

a given fluent `Fl`, the predicate `set_one_fluent` collects the list `Pos` (resp. `Neg`) of pairs `[Action,Preconditions]` such that `Action` makes `Fl` true (resp. false) in the state transition (cf. line (46)). Similarly, it handles the static causal laws (`caused` assertions), by collecting the lists of conditions that affect the truth value of `Fl` (cf. `StatPos` and `StatNeg`, in line (49)). The CLP variables involved are then constrained by the procedures `build_sum_prod` (lines (47)–(48)) and `build_sum_stat` (lines (50)–(51)). Finally, all variables related to introductions (resp. removals) of fluents are collected in `Pos_Fl` (resp. `Neg_Fl`). Their sums are collected in variables `Psum` and `Nsum`, respectively. Further constraints in `build_sum_prod` and `build_sum_stat` ensure that they are both greater than zero. Moreover, we take care of the inertia law in line (56): if none of the actions affect `Fl`, then $EV = IV$.

Let us focus on the predicate `build_sum_prod`. For the sake of simplicity, let us assume that it is called with `Mode == p` (cf. line (47)). The predicate `build_sum_prod` recursively processes a list of pairs `[Action,Preconditions]`. For each action A_j, if A_j occurs and all of its preconditions (`Prec`) hold, then there is an action effect (`Flag = 1`, line (62)). Moreover, the constraint `Mode == p -> EV #>= Flag` in line (63), imposes that, if an action A_j occurs in a state transition (i.e., the corresponding boolean variable `VA_j` is 1), and such action makes a fluent true, then the boolean variable `EV` associated to it has to be 1. Analogous constraints are imposed in the case `Mode == n`, corresponding to the handling of actions that make a given fluent false (called in line (48)).

The modus operandi for imposing the executability conditions for actions is similar. Through the predicate `set_executability` (defined in line (72)), all the preconditions of each action (cf. `Cs` in line (82)) are considered. Any boolean variable related to the occurrence of such action in a state transition is constrained with respect to the values of all variables in `Cs` (which denote the truth value of the fluents involved in the preconditions).

```
(37) set_transitions(_Occurrences,[_States]) :- !.
(38) set_transitions([O|Occurrences],[S1,S2|Rest]) :-
(39)   set_transition(O,S1,S2,S1,S2),
(40)   set_transitions(Occurrences,[S2|Rest]).
(41) set_transition(_Occ,[],[],_,_).
(42) set_transition(Occ,[fluent(F,IV)|R1],[fluent(F,EV)|R2],FromSt,ToSt):-
(43)   set_one_fluent(F,IV,EV,Occ,FromSt,ToSt),
(44)   set_transition(Occ,R1,R2,FromSt,ToSt).
(45) set_one_fluent(Fl,IV,EV,Occurrence,FromSt,ToSt) :-
(46)   findall([X,L],causes(X,Fl,L),Pos),findall([Y,M],causes(Y,mneg(Fl),M),Neg),
(47)   build_sum_prod(Pos,nOccurrence,FromSt,PFormula,EV,p),
(48)   build_sum_prod(Neg,Occurrence,FromSt,NFormula,EV,n),
(49)   findall(P,caused(P,Fl),StatPos),findall(N,caused(N,mneg(Fl)),StatNeg),
(50)   build_sum_stat(StatPos,ToSt,PStatPos,EV,p),
(51)   build_sum_stat(StatNeg,ToSt,PStatNeg,EV,n),
(52)   append(PFormula,PStatPos,Pos_Fl),
(53)   append(NFormula,PStatNeg,Neg_Fl),
(54)   sum(Pos_Fl,#=,Psum),
(55)   sum(Neg_Fl,#=,Nsum),
(56)   EV  #<=> ((Psum + IV - IV * Nsum) #> 0).
(57) build_sum_prod([],_,_,[],_,_).
(58) build_sum_prod([[Action,Prec]|Rest],Occurrence,State,[Flag|PF1],EV,Mode):-
(59)   get_precondition_vars(Prec,State,ListPV),
```

```
(60)   length(Prec,NPrec), sum(ListPV,#=,SumPrec),
(61)   member(action(Action,VA),Occurrence),
(62)   (VA #= 1 #/\ (SumPrec #= NPrec)) #<=> Flag,
(63)   (Mode == p -> EV #>= Flag; Mode == n -> EV #=< 1-Flag),
(64)   build_sum_prod(Rest,Occurrence,State,PF1,EV,Mode).
(65) build_sum_stat([],_,[],_,_).
(66) build_sum_stat([Cond|Others],State,[Flag|Fo],EV,Mode) :-
(67)   get_precondition_vars(Cond,State,List),
(68)   length(List,NL), sum(List,#=,Result),
(69)   Flag #<=> (Result #= NL),
(70)   (Mode == p -> EV #>= Flag; Mode == n -> EV #=< 1-Flag),
(71)   build_sum_stat(Others,State,Fo,EV,Mode).
(72) set_executability(ActionsOcc,States) :-
(73)   setof([Act,C],executable(Act,C),Conds),
(74)   set_executability1(ActionsOcc,States,Conds).
(75) set_executability1([],[_],_).
(76) set_executability1([AStep|ARest],[State|States],Conds) :-
(77)   set_executability_sub(AStep,State,Conds),
(78)   set_executability1(ARest,States,Conds).
(79) set_executability_sub(_Step,_State,[]).
(80) set_executability_sub(Step,State,[[Act,C]|CA]) :-
(81)   member(action(Act,VA),Step),
(82)   get_precondition_vars(C,State,Cs),
(83)   length(Cs,NCs), sum(Cs,#=,Temporary),
(84)   VA #=1 #=> Temporary #= NCs,
(85)   set_executability_sub(Step,State,CA).
(86) get_precondition_vars([],_,[]).
(87) get_precondition_vars([P1|Rest],State,[F|LR]) :-
(88)   get_precondition_vars(Rest,State,LR),
(89)   (P1 = mneg(FN),!, member(fluent(FN,A),State), F #= 1-A;
(90)    member(fluent(P1,F),State)).
(91) get_all_actions([],[]).
(92) get_all_actions([A|B],List) :-
(93)   get_action_list(A,List1),get_all_actions(B,List2),
(94)   append(List1,List2,List).
(95) get_action_list([],[]).
(96) get_action_list([action(_,V)|Rest],[V|MRest]) :-
(97)   get_action_list(Rest,MRest).
```

ASP

There are several ways to encode a block world in ASP (e.g., [22, 2]). As mentioned, we developed a translator from our action description language to *lparse*'s syntax. Such a translation is quite simple and we do not report its products. It suffices to say that much of the translation amounts to syntactical rewriting of the action theory specification to suitable ASP rules. The translator can be found in [9], together with the action theory specifications we used.

Results

Table 7 reports the execution times from the three systems, for different numbers of blocks and plan lengths (i.e., the number of moves). We also experimented with a variant of the problem described above, where a further constraint is imposed: no block x can be placed on top of another block y if $y \geq x$. This constraint can be easily

Instance (using `(6)`)		Plan	*lparse*	SMODELS	CMODELS		SICStus
Blocks	Length	exists			mChaff	Simo	
5	5	N	2.31	0.14	0.02	0.02	0.20
5	6	N	2.29	0.17	0.11	0.06	0.11
5	7	Y	2.34	0.21	0.12	0.10	0.08
6	7	N	7.64	0.32	0.16	0.13	0.31
6	8	N	7.65	0.37	0.19	0.15	1.70
6	9	Y	7.69	0.55	0.27	0.43	0.99
7	9	N	22.96	0.64	0.32	0.27	6.23
7	10	N	23.06	0.75	0.39	0.32	38.24
7	11	Y	23.10	2.15	0.57	1.35	17.40
8	11	N	36.71	1.18	0.63	0.53	154.96
8	12	N	36.81	1.92	0.74	0.62	948.31
8	13	Y	37.10	7.98	2.14	10.36	422.51
9	13	N	98.69	2.25	1.09	0.93	–
9	14	N	98.45	5.99	1.46	1.13	–
9	15	Y	100.01	433.28	4.16	23.07	–
Instance (using `(6')`)		Plan	*lparse*	SMODELS	CMODELS		SICStus
Blocks	Length	exists			mChaff	Simo	
4	4	N	0.40	0.05	$<$0.01	$<$0.01	0.08
4	5	N	0.41	0.06	0.06	0.02	0.01
4	6	Y	0.42	0.06	0.06	0.02	0.01
5	11	N	1.67	0.35	0.18	0.33	2.17
5	12	N	1.68	0.59	0.26	0.64	5.92
5	13	Y	1.68	0.75	0.28	0.50	8.07
6	25	N	3.38	–	685.43	–	–
6	26	N	3.38	–	1173.55	–	–
6	27	Y	3.41	–	1181.99	–	–

Table 7: Planning in blocks world (20 minutes time limit)

modeled in the action language, by replacing line `(6)` of the action theory specification seen above, with the following one:

```
(6') action(move(X,Y)):- blk(X), blk(Y), X>Y.
```

The results reported in Table 7 show that the ASP solvers can solve the instances (save for the easiest ones) more quickly than CLP(FD). Notice that, the CLP(FD) solution is general and applicable to every action theory described in the proposed action language. We have also tested it on planning problems for elevators, obtaining similar results. Observe also that, in CLP(FD), a price is paid in making use of a declarative translation from a generic action theory to constraints; for example, a manual and ad-hoc encoding of the same planning problem presented here (using rule `(6')`) for 5 blocks and 13 actions founds the solution in 2ms (see [9] for this code and related running times). We expect similar improvements also from an ad-hoc encoding of planning problems using ASP. Let us observe that in this problem, the time spent by lparse is not completely negligible: this is also due to the size of the file generated.

9 Knapsack

In this section, we discuss a generalization of the knapsack problem. Let us consider n types of objects, and each object of type i has size w_i and cost c_i. We wish to fill a knapsack with X_1 objects of type 1, X_2 objects of type 2, and so on, in such a way

that:

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

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

CLP(FD) Encoding

We represent the types of objects using two lists, containing the size and cost of each type of object. For instance:

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

The encoding of the problem in CLP(FD) is as follows:

```
(1)  knapsack(Max_Size,Min_Profit) :-
(2)      objects(Sizes,Costs), length(Sizes,N),
(3)      length(Vars,N),
(4)      domain(Vars,0,Max_Size),
(5)      scalar_product(Sizes,Vars,#=<,Max_Size),
(6)      scalar_product(Costs,Vars,#>=,Min_Profit),
(7)      labeling([ff],Vars).
```

Line (2) retrieves the input data and computes its length `N`. Lines (3) and (4) introduce the list of variables and fix their domains. Lines (5) and (6) encode the constraint (*). This is accomplished by using the built-in predicate

`scalar_product([`c_1`,...,`c_n`], [`V_1`,...,`V_n`],` op`, Value).`

Such predicate imposes the constraint: $c_1V_1+\cdots+c_nV_n$ op `Value` where op is a binary, finite-domain predicate such as #=<, #>=, #=, etc. [1]

ASP Encoding

Each instance of the knapsack problem is represented in ASP by a collection of facts of the form `item(Item,Weight,Cost)`, e.g.,

```
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].
```

[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.

Instance	max_size	min_profit	Answer	*lparse*	SMODELS	SICStus	GNU-Prolog
k1	255	374	Y	0.01	0.04	0.02	<0.01
k2	255	375	N	<0.01	3.08	0.03	<0.01
k3	511	757	Y	0.01	0.12	0.36	0.08
k4	511	758	N	0.02	130.82	0.36	0.08
k5	1023	1524	Y	0.03	0.49	8.81	1.92
k6	1023	1525	N	0.03	–	8.75	1.87
k7	2047	3059	Y	0.07	1.84	368.50	80.68
k8	2047	3060	N	0.08	–	366.79	80.40

Table 8: Knapsack instances (30 minute time limit)

```
(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) 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. Notice that, in CLP(FD), the same effect is obtained by bound-consistency. Rules (4)–(7) impose the constraints about minimum profit and maximum size. The two constants `max_size` and `min_profit` must be provided to *lparse* during grounding.

Results

Table 8 reports some of the results we obtained with different solvers. Notice that CMODELS seems unable to properly deal with this encoding: for all the instances we experimented with (except the smallest ones, involving at most five types of objects), the corresponding process was terminated by the O.S. The reason for this could be found by observing that the run-time images of such processes grew very large in size (up to several GBs, in some instances). We have also experimented with an alternative encoding of this problem, which uses *lparse*'s `#weight` declarations, but the encoding presented here turned out to be faster and less sensible to the numbers sizes.

The results denote a fairly irregular behavior. SMODELS has a good performance for those instances that have a solution, while it is significantly slower than the CLP(FD) solvers for instances that require the full exploration of the search space. Also in this case, SICStus is outperformed by GNU-Prolog.

10 Codes

In this section we face the problem of finding binary codes of N-elements. Let us recall some standard information-theory notions. Given two N-bit vectors $u = u_1 u_2 \dots u_N$ and $v = v_1 v_2 \dots v_N$, their *Hamming distance* is defined as $d_H(u, v) = \sum_{i=1}^{N} |u_i - v_i|$. A *code* $\mathcal{C}(N, D)$ is a subset of $\{0, 1\}^N$ such that for all words $u, v \in \mathcal{C}(N, D)$, with $u \neq v$, it holds that $d_H(u, v) \geq D$. Without loss of generality, we can assume $(0, 0, \dots, 0) \in \mathcal{C}(N, D)$.

We focus on the following decision problem: given the natural numbers N, D, M, is there a code $\mathcal{C}(N, D)$ such that $|\mathcal{C}(N, D)| = M$?

CLP(FD) Encoding

The following is our CLP-encoding of the problem:

```
 (1) code(N,D,M,Code) :-
 (2)     L is N*M, length(Code,L), domain(Code,0,1),
 (3)     constraints(Code,D,N),
 (4)     first_word(N,Code),
 (5)     lexicographic(Code,N),
 (6)     first_column(Code,N,M),
 (7)     labeling([ff],Code).
 (8) constraints(Code,_,N) :-
 (9)     length(Code,CL), CL =< N, !.
(10) constraints(Code,D,N) :-
(11)     extract(N,Code,W1,RestCode),
(12)     allpairs(RestCode,W1,D,N), constraints(RestCode,D,N).
(13) allpairs([],_,_,_).
(14) allpairs(Code,Current,D,N) :-
(15)     extract(N,Code,First,Last),
(16)     pair(Current,First,D), allpairs(Last,Current,D,N).
(17) pair(Word1,Word2,D) :-
(18)     diffvars(Word1,Word2,Diff),
(19)     domain(Diff,0,1), sum(Diff,'#>=',D).
(20) diffvars([],[],[]).
(21) diffvars([A|R],[B|S],[D|T]) :-
(22)     D #= abs(A-B), diffvars(R,S,T).
(23) first_word(N,Code) :-
(24)     extract(N,Code,W1,_),
(25)     set_bits(W1,0).
(26) lexicographic(Code,N) :-
(27)     length(Code,L), L < 2*N, !.
(28) lexicographic(Code,N) :-
(29)     extract(N,Code,W1,RestCode),
(30)     extract(N,RestCode,W2,_),
(31)     lexico(W1,W2,1), lexicographic(RestCode,N).
(32) lexico([],[],1).
(33) lexico([A|R],[B|S],C) :-
(34)     lexico(R,S,Val),
(35)     (A #< B #\/ (A #= B #/\ Val)) #<=> C.
(36) first_column(Code,N,K) :-
(37)     columns(Code,1,N,C1,_),
(38)     K1 is K // 2 + K mod 2,
(39)     extract(K1,C1,First,Last),
(40)     set_bits(First,0), set_bits(Last,1).
(41) columns([],_,_,[],[]).
(42) columns(Code,M,N,[A|R],[B|S]) :-
(43)     M1 is M+1, element(M,Code,A), element(M1,Code,B),
```

Input (n, d, k)	Exists code?	*lparse*	SMODELS		CMODELS				SICStus				GNU-Prolog			
					mChaff		Simo									
			(a)	(b)	(a)	(b)	(a)	(b)	(a)	(b)	(c)	(d)	(a)	(b)	(c)	(d)
(6,3,8)	Y	<0.01	0.02	0.02	0.08	0.10	0.05	0.08	<0.01	0.01	<0.01	<0.01	<0.01	<0.01	<0.01	<0.01
(6,3,9)	N	<0.01	0.02	0.02	0.13	0.20	0.13	0.18	573.46	0.08	2.26	0.01	54.04	0.01	0.24	<0.01
(7,3,16)	Y	0.01	0.09	0.09	1.19	2.05	2.83	9.40	0.04	0.01	0.01	0.02	<0.01	<0.01	<0.01	<0.01
(7,3,17)	N	0.01	6.11	6.92	2.74	7.36	17.51	20.97	–	865.50	–	3.30	–	118.81	–	0.42
(8,3,20)	Y	0.04	0.22	224.37	7.75	1188.34	223.91	–	–	–	–	–	–	–	–	–
(8,3,21)	?	0.04	–	–	–	–	–	–	–	–	–	–	–	–	–	–
(8,5,4)	Y	0.07	0.11	0.11	0.10	0.12	0.07	0.10	<0.01	<0.01	0.01	<0.01	<0.01	<0.01	<0.01	<0.01
(8,5,5)	N	0.07	0.40	0.41	0.36	0.51	0.42	0.78	3.90	0.02	0.42	0.01	0.41	<0.01	0.04	<0.01
(9,3,32)	Y	0.10	0.38	626.18	16.90	10.24	–	–	0.65	0.34	0.09	0.07	0.11	0.04	0.03	<0.01
(9,3,33)	Y	0.10	388.80	–	6.71	11.86	–	–	–	–	–	–	–	–	–	1931.52
(9,5,6)	Y	0.31	24.33	24.64	0.49	0.46	2.25	4.04	0.02	<0.01	0.01	0.01	<0.01	<0.01	<0.01	<0.01
(9,5,7)	N	0.31	80.15	81.44	124.48	144.39	267.00	372.06	–	7.79	252.95	1.70	–	1.10	38.99	0.20
(10,5,12)	Y	1.24	–	–	82.85	205.45	–	–	792.78	0.04	14.87	0.03	89.76	<0.01	2.50	<0.01
(10,5,13)	N	1.24	–	–	–	–	–	–	–	–	–	374.39	–	–	–	43.28

Table 9: The codes problem (30 minutes time limit)

```
(44)     extract(N,Code,_,RestCode), columns(RestCode,M,N,R,S).
(45) extract(N,Code,First,Last) :-
(46)     length(First,N), append(First,Last,Code).
(47) set_bits([],_).
(48) set_bits([N|R],N) :- set_bits(R,N).
```

Line (2) introduces `M`×`N` binary variables, i.e., `N` of them for each word. The predicate `constraint` (line (3)) introduces the distance constraints among words. Each pair of words is selected in lines (8)–(16), using a recursive procedure. For each pair of words, the list `Diff` of difference variables is generated (lines (17)–(19)), and the number of difference variables that are different from 0 is set to be at least `D` in line (19). The auxiliary predicate `extract(N,List, First,Last)`, defined in lines (45)–(46), is used to split a list `List` into a prefix `First` of `N` elements and the remaining part `Last`. We employ three simple constraint optimizations:

- The predicate `first_word` sets the first word to be $(0, 0, \ldots, 0)$.
- The predicate `first_column` requires the first half of the words to start with 0 and the remaining ones with 1.
- The predicate `lexicographic` requires that the code is computed using lexicographic ordering among tuples, to reduce the number of symmetries.

Lexicographic ordering is imposed by using reified constraints (line (35)). We also experimented with imposing lexicographic ordering by using a binary/decimal conversion (i.e., $(X_{n-1}, X_{n-2}, \ldots, X_1, X_0)$ is considered as $2^{n-1}X_{n-1}+2^{n-2}X_{n-2}+\cdots+2^1X_1+X_0$ and those values are constrained to be strictly increasing), but the resulting solution provided slightly worse performance.

ASP Encoding

The key idea in obtaining an ASP-encoding of the problem is that a word u belongs to the code if (and only if) all other words of distance less than d from u do not belong to the code (an analogous encoding is used in [31]). For instance, for $n = 4, d = 3$ the

following program can answer the question $|\mathcal{C}(n,d)| = m$? for any m:

```
 (1) bit(0).   bit(1).
 (2) word(0, 0, 0, 0).
 (3) word(Var4,Var3,Var2,Var1) :-
 (4)   not word(Var4,Var3,Var6,Var5),
 (5)   not word(Var4,Var7,Var2,Var5),
 (6)   not word(Var8,Var3,Var2,Var5),
 (7)   not word(Var4,Var7,Var6,Var1),
 (8)   not word(Var8,Var3,Var6,Var1),
 (9)   not word(Var8,Var7,Var2,Var1),
(10)   not word(Var4,Var3,Var2,Var5),
(11)   not word(Var4,Var3,Var6,Var1),
(12)   not word(Var4,Var7,Var2,Var1),
(13)   not word(Var8,Var3,Var2,Var1),
(14)   bit(Var4),bit(Var8),Var4 != Var8 ,
(15)   bit(Var3),bit(Var7),Var3 != Var7 ,
(16)   bit(Var2),bit(Var6),Var2 != Var6 ,
(17)   bit(Var1),bit(Var5),Var1 != Var5.
(18) :- (m/2+(m mod 2)+1) {
         word(0,Var8,Var7,Var6,Var5,Var4,Var3,Var2,Var1) :
         bit(Var8;Var7;Var6;Var5;Var4;Var3;Var2;Var1)} m.
(19) :- (m/2+1) {word(1,Var8,Var7,Var6,Var5,Var4,Var3,Var2,Var1) :
         bit(Var8;Var7;Var6;Var5;Var4;Var3;Var2;Var1)} m.
(20) solutions :- m {word(Var4,Var3,Var2,Var1) :
                bit( Var4;Var3;Var2;Var1)} m.
(21) :- not solutions.
```

We uniformly generate programs of this form, for various values of n and d, by means of a Prolog program—such Prolog program that can be found in [9]. Let us describe the above ASP program. The domains are set in line (1). The fact (2) imposes that $(0,0,\ldots,0)$ belongs to the code (similarly to what done in CLP(FD)). The rules in lines (3)-(17) impose the constraints on the distance among words. Let us observe that the number of atoms of the form `not word(...)` depends on `N` and `D`. In general, their number is $\sum_{i=1}^{D-1}\binom{n}{i}$. For the particular case at hand, lines (4)–(9) handle tuples of distance 2, while lines (10)–(13) are related to tuples of distance 1. The constraints (18)–(19) impose that half of the words begin with 0. If `m` is odd, the number of words starting with 0 is greater than those starting with 1. Lines (20) and (21) require `m` words in the solution, where `m` is a constant to be supplied at grounding time.

Results

The results are reported in Table 9; the table describes four classes of experiments, depending on which of the three constraints mentioned above have been allowed in the program. In particular,

(a) the word $(0,0,\ldots,0)$ is forced to belongs to the code;
(b) as (a) and half of the words must begin with 0;

(c) as (a) but lexicographic ordering among tuples is imposed;
(d) as both (b) and (c).

Notice that the ASP code for (b), after grounding, is more complex that the one for (a). This translates to worse performance, as shown in Table 9. Moreover, in ASP it is not natural to impose lexicographical ordering on the words of the solutions—an answer set is a set and it can be computed in any order. One could introduce a predicate `ord_word`, of arity $n+1$, and impose that if for $i < j$ both `ord_word(i,x1,...,xn)` and `ord_word(j,y1,...,yn)` hold, then `(x1,...,xn)` precedes lexicographically `(y1,...,yn)`. This leads to a complex and rather non-declarative encoding. We do not report on these experiments.

11 Space consumption

Execution time is not the only aspect to be considered in comparing different approaches and solutions to problems. Some relevance should to be given also to other factors, such as memory requirements, the amount of time needed to develop a program, and the level of expertise the programmer must have in order to suitably master a specific declarative framework. In this section, we focus on the first of these aspects. The other aspects could be clearly related to the "declarative level" of the framework, and will be briefly discussed in the concluding section.

The memory amounts reported in Table 10 correspond to the maximum amount of virtual memory used by the ASP solvers in a selection of the experiments described in this paper. The memory consumption includes all code, data, and shared libraries plus pages that have been swapped out. These data have been obtained through O.S. administrator tools. As concerns the runs of CMODELS using mChaff as SAT-solver, we report both the maximum amount of memory used by CMODELS and by mChaff separately. This is possible because CMODELS runs mChaff as an independent process, while this is not the case for Simo. Hence, for CMODELS+Simo we report only the overall amount of memory required.

It should be noticed that in many of the cases in which the ASP solvers did not produced an answer, within a reasonable time (cf. Tables 1-4,7,9), the run-time images of the process grew very large in size (much larger that the amount of available RAM). As a result, most of the time is spent in swapping pages in and out, slowing down the overall computation.

Regarding the CLP(FD) solvers, both SICStus Prolog and GNU-Prolog (and similar argument applies to other solvers), adopt more efficient memory management schemes, including the use of an automated garbage collector. From the O.S., we gathered the data regarding the dimension of SICStus and GNU-Prolog running processes: in our experimentation, SICStus always required about 260MB, while all runs of GNU-Prolog were completed by allocating at most 32MB.

	Instance	SMODELS	CMODELS	
			mChaff	Simo
3-coloring	4-FullIns_5	49	70 (38)	103
	3-Insertions_3	27	3 (4)	14
	DSJR500.5	34	48 (23)	71
	DSJC500.5	36	51 (24)	75
	gen400_p0.9_55	41	57 (34)	86
	gen400_p0.9_65	41	57 (30)	85
	wap05a	26	37 (20)	55
	wap06a	27	38 (18)	55
	wap07a	59	84 (38)	122
	wap08a	60	85 (41)	123
4-coloring	3-FullIns_5	27	38 (21)	57
	4-FullIns_5	58	83 (40)	121
	DSJR500.5	41	55 (32)	81
	DSJC500.5	43	59 (29)	86
	gen400_p0.9_55	49	67 (31)	97
	gen400_p0.9_65	49	67 (35)	97
	wap05a	31	43 (18)	63
	wap06a	31	44 (25)	64
	wap07a	71	98 (61)	141
	wap08a	71	99 (61)	143
5-coloring	DSJR500.5	47	63 (34)	92
	DSJC500.5	50	67 (36)	98
	gen400_p0.9_55	56	76 (45)	108
	gen400_p0.9_65	56	76 (45)	108
	wap05a	36	49 (31)	72
	wap06a	36	50 (29)	73
	wap07a	82	109 (64)	162
	wap08a	82	110 (65)	163
Schur	$\langle 4, 45\rangle$	4	4 (76)	143
	$\langle 4, 46\rangle$	4	4 (73)	122
	$\langle 4, 47\rangle$	4	4 (70)	141
	$\langle 4, 48\rangle$	4	4 (70)	145
	$\langle 4, 49\rangle$	4	4 (87)	125

	Instance	SMODELS	CMODELS	
			mChaff	Simo
Queens	$M = N = 6$	3	4 (36)	40
	$M = 6, N = 7$	3	4 (3)	6
	$M = 9, N = 8$	4	6 (8)	–
Hamiltonian	hc1	12	23 (15)	26
	hc2	12	158 (100)	26
	hc3	12	20 (13)	26
	hc4	12	24 (23)	26
	np50c	4	43 (47)	28
	np60c	4	35 (30)	45
	np70c	5	45 (41)	70
	np80c	6	64 (56)	98
	np90c	7	102 (86)	145
HP-Decision	$E(h^{15})$	5	13 (17)	27
	$E(h^{20})$	11	37 (114)	229
	$E(h^{25})$	22	90 (433)	–
	$E((hpp)^5h)$	8	23 (31)	68
	$E((hpp)^7h)$	12	41 (217)	112
	$E((hpp)^9h)$	26	30 (438)	295
Planning	$\langle 7, 9\rangle$	6	10 (7)	14
	$\langle 7, 10\rangle$	6	11 (7)	16
	$\langle 7, 11\rangle$	7	12 (9)	19
	$\langle 8, 11\rangle$	8	16 (11)	24
	$\langle 8, 12\rangle$	9	18 (12)	27
	$\langle 8, 13\rangle$	9	20 (16)	35
Codes	(7,3,17)(a)	3	6 (8)	23
	(7,3,17)(b)	3	9 (9)	21
	(8,3,20)(a)	3	12 (15)	94
	(8,3,20)(b)	3	19 (170)	–
	(9,5,6)(a)	3	10 (8)	17
	(9,5,6)(b)	3	13 (10)	23
	(9,5,7)(a)	3	11 (39)	64
	(9,5,7)(b)	3	14 (42)	74

Table 10: Space consumption of ASP solvers (MBs)

12 Discussion

We tested the CLP(FD) and ASP codes for various combinatorial problems. In the Tables 1–9 we reported the running times (in seconds) of the solutions to these problems on different problem instances, while in Section 11 we gathered an excerpt of the data regarding memory consumption. Let us try to analyze these results.

12.1 Comparing CLP(FD) and ASP

First of all, it is clear that ASP provides a more compact, and probably more declarative, encoding. In particular, the use of grounding and domain-restricted variables allows ASP programmers to avoid the explicit use of recursion in most situations. In general, in designing the CLP(FD) code, the programmer cannot easily ignore the specific inference strategy adopted by the CLP(FD) engine. The fact that CLP(FD) adopts a top-down, depth-first strategy affects the programmer's choices in encoding the algorithms. As a result, the amount of effort required to acquire good programming skills in the two paradigms is very different. Indeed, teaching experiences with undergraduate students confirm that ASP is easier to learn, especially without previous knowledge of

declarative programming.

On the other hand, it is reasonable that deeper understanding of specific "non-declarative" aspects of the CLP(FD) framework allows the programmer to develop more efficient, albeit more complex, solutions. The problem of code generation (Section 10) is paradigmatic: the more the CLP(FD) solution is refined (by introducing further heuristics) the faster is the execution. Conversely, attempts to surrogate the same heuristics in the profoundly declarative framework of ASP yielded (in our experiments) unsatisfactory results.

As far as running times are concerned, CLP(FD), in most of the cases, 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. Nevertheless, there are cases (e.g., the code-generation problem) in which CLP(FD) fails in solving instances that are relatively easy for the ASP solvers. 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 [12], 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 [39]).

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 CLP(FD) performs better. On the other hand, whenever there is no solution, running times are favorable to CMODELS. 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). The same behavior of CMODELS can be also observed in other situations (see the Hamiltonian circuit problem). 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. A further challenging theme for future research is to understand how such intrinsic structure of a problem could be exploited in order to better drive the interaction with the SAT-solver (especially for non-tight problems).

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

Table 11: Schematic results' analysis for execution time. + (-) means that the formalism is (not) applicable. ++ that it is the best when the two formalisms are applicable.

Regarding the protein folding problem, CLP(FD) solves the optimization problems much faster than ASP. In the decision version, times are quite close. Also in this case, however, the ASP code appears to be simpler and more compact than the CLP(FD) one.

For the planning problem, we observe that SMODELS has running times higher but usually comparable with CMODELS. The CLP(FD) approach, although correct, seems less suitable to this problem, especially when the size of the instance grows.

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

Table 12.1 intuitively summarizes our observations drawn from the different benchmarks. From this experience we can draw 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 and 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., [13, 34, 8]). Nevertheless, we tested the CLP(FD) programs using SICStus Prolog, B-Prolog, ECLiPSe, and GNU-Prolog [37]. As far as running times are concerned, such partial experimentation indicates that B-Prolog and SICStus Prolog have comparable behavior (with B-Prolog being a bit faster), GNU-Prolog is the fastest, and ECLiPSe is the slowest.

In our experiments, we have isolated the grounding phase from the answer set computation timings, and reported them separately. Nevertheless, in our tests, the time spent in grounding is often negligible (see the case of planning in Section 8 for the unique exception) w.r.t. SMODELS/CMODELS running time, save for the easier instances. Regarding the use of different SAT-solvers with CMODELS (we also tested zChaff and RelSat), as one can expect (see, e.g., [17]) the choice of the SAT solver affects performance, but there is not a definite winner.

12.2 ASP vs. SAT

As mentioned, our experiments suggest that the greater efficiency of CMODELS, with respect to SMODELS, originates from the high speed of the underlying SAT solvers. In

Instance		CMODELS			Sat solver		
Graph	Colors	mChaff	Simo	zChaff	mChaff	Simo	zChaff
3-FullIns_5	3	1.23	1.18	1.17	1.04	0.08	0.01
4-FullIns_5	3	2.69	2.69	2.65	2.41	0.28	0.02
2-Insertions_4	3	0.15	1.08	0.13	0.13	59.54	0.08
2-Insertions_5	3	0.50	2.62	0.69	0.33	–	0.18
3-Insertions_3	3	4.16	7.59	2.64	5.22	61.67	–
4-Insertions_3	3	1772.14	1481.27	213.22	–	–	–
4-Insertions_4	3	1443.33	1468.91	–	876.77	–	–
DSJR500.5	3	1.81	1.78	1.78	1.60	0.01	0.01
DSJC500.5	3	1.92	1.92	1.87	1.70	0.01	0.01
gen400_p0.9_55	3	2.19	2.18	2.15	1.91	0.01	0.01
gen400_p0.9_65	3	3.16	2.15	2.17	1.92	0.01	0.01
le450_5a	3	0.24	0.21	0.20	0.17	0.01	<0.01
le450_5b	3	0.24	0.20	0.20	0.17	<0.01	<0.01
le450_5c	3	0.37	0.33	0.34	0.28	<0.01	<0.01
le450_5d	3	0.36	0.33	0.33	0.27	<0.01	<0.01
wap05a	3	1.38	1.36	1.36	1.27	<0.01	0.01
wap06a	3	1.42	1.40	1.34	1.27	<0.01	0.01
wap07a	3	3.28	3.31	3.28	3.04	0.01	0.01
wap08a	3	3.31	3.32	3.32	3.07	0.01	0.01
3-FullIns_5	4	1.51	1.47	1.47	1.48	0.25	0.03
4-FullIns_5	4	3.37	3.38	3.33	3.41	0.31	0.06
2-Insertions_4	4	–	–	–	–	–	–
2-Insertions_5	4	–	–	–	–	–	–
3-Insertions_3	4	0.04	0.01	0.01	0.01	<0.01	<0.01
4-Insertions_3	4	0.04	0.01	0.01	0.01	<0.01	<0.01
4-Insertions_4	4	–	–	–	–	–	–
DSJR500.5	4	2.26	2.19	2.19	2.17	0.10	0.02
DSJC500.5	4	2.36	2.37	2.29	2.30	0.34	0.02
gen400_p0.9_55	4	2.68	2.69	2.66	2.66	0.05	0.02
gen400_p0.9_65	4	2.67	2.66	2.67	2.65	0.06	0.02
le450_5a	4	0.29	0.27	0.26	0.24	1.50	<0.01
le450_5b	4	0.29	0.27	0.26	0.24	1.33	0.01
le450_5c	4	0.46	0.45	0.42	0.40	1.67	0.01
le450_5d	4	0.44	0.42	0.42	0.39	8.60	0.01
wap05a	4	1.73	1.70	1.71	1.71	0.08	0.01
wap06a	4	1.75	1.70	1.71	1.74	0.10	0.01
wap07a	4	4.12	4.09	4.08	4.08	0.22	0.02
wap08a	4	4.15	4.19	4.15	4.14	0.24	0.02
3-FullIns_5	5	1.69	2.04	1.91	1.97	442.03	0.13
4-FullIns_5	5	4.19	4.13	4.11	4.58	1.16	0.18
2-Insertions_4	5	0.07	0.04	0.04	0.04	<0.01	<0.01
2-Insertions_5	5	–	–	–	–	–	–
3-Insertions_3	5	0.04	0.01	0.01	0.01	<0.01	<0.01
4-Insertions_3	5	0.05	0.02	0.01	0.01	<0.01	<0.01
4-Insertions_4	5	0.32	0.35	0.15	0.14	<0.01	<0.01
DSJR500.5	5	2.71	2.68	2.65	2.79	5.36	0.06
DSJC500.5	5	2.84	2.99	2.84	2.96	1567.77	0.07
gen400_p0.9_55	5	3.24	3.22	3.18	3.40	1.70	0.05
gen400_p0.9_65	5	3.22	3.23	3.17	3.37	1.89	0.06
le450_5a	5	12.30	1.99	–	0.90	–	0.28
le450_5b	5	0.98	12.47	–	2.66	–	–
le450_5c	5	0.70	0.58	0.51	0.52	–	0.01
le450_5d	5	0.60	0.51	–	0.59	1021.36	0.30
wap05a	5	2.07	2.05	2.04	2.17	1.13	0.04
wap06a	5	2.13	2.10	2.09	2.22	2.56	0.05
wap07a	5	4.99	5.05	4.92	5.21	13.90	0.07
wap08a	5	5.08	5.00	4.97	5.29	12.21	0.11

Table 12: Graph k-coloring: ASP vs SAT (30 minutes time limit)

view of this fact, one could wonder whether it is better to solve an instance of a hard combinatorial problem by using a SAT-based ASP solver or by directly employing a stand-alone SAT solver. The two alternatives require different approaches in modeling the problem, and different amounts of effort in encoding the problem. A comparison between SAT and ASP solvers is important, but goes beyond the scope of this article. Nevertheless, we made a first step in this direction by comparing the behavior of CMODELS with the stand-alone implementations of the same SAT solvers it uses, namely zChaff, Simo, and mChaff [38].

The experiments we performed make use of a selection of instances of two specific problems, namely graph k-coloring and Hamiltonian circuit. A motivation for this choice is that, in our encodings, such problems lead to tight and non-tight programs, respectively. The instances to be processed by the SAT solvers have been encoded in the DIMACS format, and generated by applying two well-known translations of Graph coloring (e.g., [30]—Section 1.1) and Hamiltonian circuit (e.g., [29]—Example 8.1) instances to SAT.

As shown in Tables 12 and 13, the tightness of programs significantly affects the behavior of the solvers. On one hand, whenever a tight instance is tackled, it is often the case that the SAT solvers outperform CMODELS. On the other hand, very often non-tight programs are efficiently solved by the ASP solver while all SAT solvers fail.

Such failures are a direct consequence of the growth in the size of the run-time image of the processes. We have already observed how this phenomenon also affects CMODELS when processing the instances of the Knapsack problem (see Section 9), which actually involves tight instances.

The better behavior of CMODELS in dealing with non-tight programs could be intuitively explained by observing that, while a SAT solver tries to solve the entire instance, CMODELS exploits a SAT solver to drive the construction of the answer sets through the discovery of loop formulae. This use of a SAT solver usually involves smaller SAT instances.

13 Conclusions

In this paper, we described an experimental study aimed at comparing the performance of CLP(FD) and ASP on various classes of combinatorial problems. The aim of the study is to provide a better understanding of what makes one paradigm more suitable than the other in solving combinatorial problems. We provided what we feel are the most natural and declarative encodings of the various problems in the different paradigms, and we analyzed the performance figures obtained by running such instances through different CLP(FD) and ASP solvers.

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). We are interested in answering the following questions:

- Is it possible to formalize the domain and problem characteristics and employ this to choose 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)?

Instance Graph	CMODELS mChaff		Simo		zChaff		Sat solver mChaff	Simo	zChaff
hc1	37.60	(56)	14.30	(33)	5.53	(10)	–	–	–
hc2	1390.15	(681)	3.48	(5)	3.25	(5)	–	–	–
hc3	20.08	(33)	6.52	(14)	10.71	(25)	–	–	–
hc4	92.99	(117)	6.54	(13)	4.11	(8)	–	–	–
hc5	0.22	(0)	0.21	(0)	0.21	(0)	123.21	–	0.60
hc6	0.21	(0)	0.21	(0)	0.21	(0)	123.52	–	0.58
hc7	0.21	(0)	0.21	(0)	0.21	(0)	123.32	–	0.45
hc8	0.21	(0)	0.21	(0)	0.21	(0)	123.69	–	0.44
np10c	0.04	(0)	0.01	(4)	0.01	(4)	<0.01	0.01	<0.01
np20c	0.81	(17)	0.05	(11)	0.07	(9)	0.04	–	<0.01
np30c	0.26	(1)	0.20	(24)	0.27	(14)	0.12	–	<0.01
np40c	4.35	(14)	0.68	(37)	0.76	(19)	–	–	<0.01
np50c	117.77	(115)	1.55	(51)	1.72	(24)	–	–	<0.01
np60c	24.72	(27)	5.54	(136)	3.77	(29)	–	–	0.01
np70c	9.46	(7)	7.46	(118)	5.91	(34)	–	–	0.01
np80c	12.53	(6)	12.60	(152)	9.20	(39)	–	–	0.01
np90c	127.31	(43)	42.32	(411)	14.24	(44)	–	–	0.01
2xp30	0.02	(0)	0.02	(0)	0.02	(0)	–	7309.64	–
2xp30.1	4.59	(51)	377.09	(127)	371.81	(23)	–	–	–
2xp30.2	2.68	(43)	3.90	(251)	21.47	(50)	–	–	–
2xp30.3	2.69	(43)	3.87	(251)	21.49	(50)	–	–	–
2xp30.4	5692.11	(26)	–		553.54	(44)	–	–	–
4xp20	0.04	(0)	0.04	(0)	0.04	(0)	3008.58	–	–
4xp20.1	1.48	(1)	36.36	(1)	3.56	(1)	4754.89	–	–
4xp20.2	3.32	(43)	3.42	(108)	7.43	(99)	–	–	–
4xp20.3	2.62	(5)	51.45	(12)	1.46	(10)	–	–	–

Table 13: Hamiltonian circuit: ASP vs SAT (180 minutes time limit)

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

Acknowledgments

We thank Yuliya Lierler, Marco Maratea, Tran Cao Son, and Neng-Fa Zhou for useful discussions on the topics of this paper. This work is partially supported by MIUR-PRIN project n.2005-015491, GNCS2005 project on constraints and their applications, and by NSF grants CNS-0220590, CNS-0454066, and HRD-0420407.

References

[1] C. Anger, T. Schaub, 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, D. Goldman, C. H. Papadimitriou, A. Piccolboni, and M. Yannakakis. On the complexity of protein folding. *J. of Computational Biology*, 5(3):423–466, 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] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[7] M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *J. of the ACM*, 7(3):201–215, 1960.

[8] D. Diaz and P. Codognet. Design and Implementation of the GNU-Prolog System. *J. of Functional and Logic Programming*, 2001(6), 2001.

[9] A. Dovier, A. Formisano, and E. Pontelli. A comparison of CLP(FD) and ASP in tackling hard combinatorial problems. Web site: `www.di.univaq.it/~formisano/CLPASP`.

[10] I. Elkabani, E. Pontelli, and T. C. Son. Smodels with CLP and Its Applications: A Simple and Effective Approach to Aggregates in ASP. In Proc. of *ICLP04*, Springer Verlag, pp. 73–89, 2004.

[11] O. Elkhatib, E. Pontelli, and T. C. Son. A Tool for Reasoning about Answer Sets in Prolog. In Proc. of *PADL*, Springer Verlag, pp. 148–162, 2004.

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

[13] 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.

[14] M. Gelfond. Representing Knowledge in A-Prolog. In *Computational Logic: Logic Programming and Beyond*, Springer Verlag, pp. 413–451, 2002.

[15] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In Proc. of *ICLP88*, pp. 1070–1080, MIT Press, 1988.

[16] M. Gelfond and V. Lifschitz. Action Languages. *Electron. Trans. Artif. Intell.*, 2:193–210, 1998.

[17] E. Giunchiglia, Y. Lierler, and M. Maratea. SAT-Based Answer Set Programming. In Proc. of *AAAI'04*, pp. 61–66, AAAI/Mit Press, 2004.

[18] E. Giunchiglia, Y. Lierler, and M. Maratea. Answer Set Programming based on Propositional Satisfiability. *J. Automated Reasoning*, to appear, Kluwer, 2006.

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

[20] J. Lee and V. Lifschitz. Loop formulas for disjunctive logic programs In Proc. of *ICLP03*, pp. 451–465, 2003.

[21] Y. Lierler and M. Maratea. CMODELS-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In Proc. of *LPNMR04*, pp. 346–350. Springer Verlag, 2004.

[22] V. Lifschitz. Answer Set Planning. In Proc. of *ICLP99*, MIT Press, pp. 23–37, 1999.

[23] V. Lifschitz and A. Razborov. Why are there so many loop formulas? *ACM Transactions on Computational Logic*, 7(2):261–268, 2006.

[24] F. Lin and Y. Zhao. ASSAT: Computing Answer Sets of A Logic Program By SAT Solvers. In *AAAI'02*, pp. 112–117. AAAI/MIT Press, 2002.

[25] V. W. Marek and M. Truszczyński. Autoepistemic Logic. *J. of the ACM*, 38(3):588–619, 1991.

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

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

[28] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In Proc. of *Design Automation Conference*, ACM Press, pp. 530–535, 2001.

[29] C. H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[30] S. D. Prestwich. Local Search on SAT-encoded Colouring Problems. In Proc. of *SAT 2003, 6th International Conference*, LNCS 2919, pp. 105–119, 2003.

[31] P. Simons. Extending and Implementing the Stable Model Semantics. Doctoral dissertation. Report 58, Helsinki University of Technology, 2000.

[32] T. C. Son, C. Baral, and S. McIlraith. Planning with different forms of domain-dependent control knowledge — an answer set programming approach. In Proc. of *LPNMR01*, Springer Verlag, pp. 226–239, 2001.

[33] M. Thielscher. Reasoning about Actions with CHRs and Finite Domain Constraints. In Proc. of *ICLP02*, LNCS 2401, pp. 70–84, 2002.

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

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

[36] 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.

[37] Web references for some CLP(FD) implementations. SICStus Prolog: www.sics.se/isl/sicstuswww/site/index.html. B-Prolog: www.probp.com. ECLiPSe: eclipse.crosscoreoptimization.com. GNU-Prolog: pauillac.inria.fr/~diaz/gnu-prolog.

[38] Web references for some SAT solvers. mChaff and zChaff: www.princeton.edu/~chaff/software.html. Simo: www.star.dist.unige.it/~sim/simo/.

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