Comparing ASP, CP, ILP on two Challenging Applications: Wire Routing and Haplotype Inference

Elvin Çoban, Ferhan Türe, and Esra Erdem

Sabancı University, Istanbul 34956, Turkey

Abstract. We study three declarative programming paradigms, Answer Set Programming (ASP), Constraint Programming (CP), and Integer Linear Programming (ILP), on two challenging applications: wire routing and haplotype inference. We represent these two problems in each formalism in a systematic way, compare these formulations both from the point of view of knowledge representation (e.g., how tolerant they are to elaborations) and from the point of view of computational efficiency (in terms of computation time and program size). We discuss possible ways of improving the computational efficiency, and other reformulations of the problems based on different mathematical models. This paper not only reports a comparison of ASP, CP, ILP based on our experiences, but also introduces new approaches to solve wire routing and haplotype inference.

1 Introduction

Wire routing is the problem of determining the physical locations of all the wires interconnecting the circuit components (e.g., transistors, gates, functional units) on a chip, possibly in the presence of obstacles (e.g., parts of the chip occupied by existing devices such as memory and registers). We consider in particular one sort of wire routing that asks for a configuration of wires connecting a given set of pins such that the wires do not go through obstacles, and that the total wire length be minimum (motivated by delay minimization). The decision version of this problem is NP-hard [1].

Each genotype (the specific genetic makeup of an individual) has two copies, one from the mother and one from the father. These two copies are called haplotypes, and they combine to form the genotype. Due to technological limitations, we have access to genotype data rather than haplotype data. Motivated by the goal of identifying maternal and paternal inheritance to be able to map disease genes, and find the set of genes responsible for a particular disease, researchers have been studying how to infer haplotypes that form a given set of genotypes. We consider in particular a slight modification of the haplotype inference problem that asks for a minimal set of haplotypes that explain the given genotypes; the decision version is NP-hard [2, 3].

We consider the decision problems corresponding to the problems above (since the optimization versions are too hard for the existing solvers), and represent each decision problem in three declarative programming paradigms, Answer Set Programming (ASP), Constraint Programming (CP), and Integer Linear Programming (ILP), in a systematic way: First we describe the constraints in a metalanguage, as precise and straightforward as possible, and then formulate each constraint in each formalism. We compare these formulations both from the point of view of knowledge representation (e.g., how easy it is to express some concepts and constraints, like reachability and aggregates, and how tolerant the representations are to elaborations) and from the point of view of computational efficiency (i.e., in terms of CPU time and program size). After that we discuss possible ways of improving the computational efficiency (e.g., by adding symmetry breaking constraints, introducing auxiliary variables, and adding redundant constraints) taking into account elaboration tolerance, and other reformulations of the problems based on different mathematical models. Also we consider alternative mathematical models of wire routing and haplotype inference "tuned" for one particular formalism (usually ILP), and examine corresponding reformulations in other formalisms. In our experiments, we use the ASP solvers CLASP[1] and CMODELS;[2] and the commercial CP and ILP solver ILOG OPL.[3]

One concept that we use when comparing formalizations is elaboration tolerance, introduced by John McCarthy in [4]: A formalism is elaboration tolerant if it can be easily modified to adapt to new phenomena. The simplest modification to a formulation is adding new formulas and predicates, whereas adding arguments to formulas and predicates is considered to change a formulation the most.

A similar study that compares ASP and CP/ILP is [5]: the authors compare the computation times of some solvers on some problems, relative to various reformulations (e.g., by symmetry breaking, and adding auxiliary variables). Our work can be seen as a continuation and/or a complement of this study. On the other hand, our paper not only reports a comparison of ASP, CP, ILP based on our experiences, but also introduces new approaches and formulations to solve wire routing and haplotype inference.

In the following, we give a brief summary of ASP, with some examples, and, due to space limitation, we refer the readers to relevant papers and books for more information about ASP, ILP, and CP [7–9]. All formulations of the problems mentioned in the paper, as well as results of all our experiments, are available at the web page [6].

2 Answer Set Programming

Answer Set Programming (ASP) [7] is a form of declarative programming that provides an expressive, declarative language to represent combinatorial search problems, and efficient solvers to compute solutions. In ASP the idea is to represent the given computational problem as a "program" so that the models of the program (called "answer sets" [10]) correspond to the solutions of the given problem. Answer sets for a program can be computed by "answer set solvers" such as CLASP and CMODELS.

CLASP and CMODELS have the same input language as that of the "grounder" LPARSE;[4] some parts of programs look like Prolog programs. However, their computation mechanisms are different: CLASP implements an algorithm inspired by SAT technology, whereas CMODELS transforms the ground program to a propositional theory and uses a SAT solver (in our experiments, MINISAT[5]) to compute solutions.

[1] `http://www.cs.uni-potsdam.de/clasp/`
[2] `http://www.cs.utexas.edu/users/tag/cmodels.html`
[3] `http://www.ilog.com/products/solver`
[4] `http://www.tcs.hut.fi/Software/smodels/`
[5] `http://minisat.se/`

We refer the reader to [11] for a definition of an answer set for a program, and give some examples instead in the language of LPARSE:

Program	Answer Sets
`p :- not q.`	{p}

`p :- not q.`
`q :- not p.`
	{p}, {q}

`p :- not q.`
`q :- not p.`
`r :- p.`
`r :- q.`
	{p,r}, {q,r}

`p :- not not p.`	{}, {p}

When we represent a problem in ASP, two kinds of rules play a major role: those that "generate" many answer sets corresponding to "possible solutions", and those that can be used to "weed out" the answer sets that do not correspond to solutions. For instance, the last rule above is of the former kind: it generates two answer sets. Constraints (rules with empty head) are of the latter kind. For instance, adding the constraint

```
:- p.
```

to the last program above eliminates the answer set that contains p.

In ASP, two special constructs of the forms $\{A_1, \ldots, A_n\}$ and $l\{A_1, \ldots, A_m\}u$ can be used in programs, where each A_i is an atom; the nonnegative integers l and u (the "lower bound" and the "upper bound") are optional [12]. The first expression describes subsets of $\{A_1, \ldots, A_n\}$. Such expressions can be used in heads of rules to generate many answer sets. For instance, the answer sets for the program

```
{p,q,r}.
```

are arbitrary subsets of {p,q,r}. The second expression describes the subsets of the set $\{A_1, \ldots, A_m\}$ whose cardinalities are at least l and at most u. Such expressions when used in heads of rules generate many answer sets whose cardinality is at least l and at most u, and when used in constraints eliminate some answer sets. For instance, adding to program above the constraint

```
:- 2 {p,q,r}.
```

eliminates its answer sets whose cardinalities are at least 2.

A group of rules that follow a pattern can be often described in a compact way using variables. Variables must be capitalized. For instance, we can write the program

$$p_i \leftarrow \textit{not}\ p_{i+1} \qquad (1 \leq i \leq 7)$$

as follows

```
index(1..7).
p(I) :- not p(I+1), index(I).
```

where the symbol index represents a number between 1 and 7. The auxiliary symbols used in the input language of LPARSE to describe the ranges of variables, such as index, are called domain predicates. The "definitions" of domain predicates, such as index(1..7), tell LPARSE how to substitute specific values for variables in schematic expressions, such as p(I) :- not p(I+1). Variables can be also used "locally" to describe the list of formulas. For instance, the rule $1\{p_1, \ldots, p_7\}1$ can be represented in an LPARSE input file by

```
1{p(I):index(I)}1.
```

3 Wire Routing Problems

Wire routing is the problem of determining the physical locations of all the wires interconnecting the circuit components (e.g., transistors, gates, functional units) on a chip, possibly in the presence of obstacles (e.g., parts of the chip occupied by existing devices such as memory and registers). We consider in particular one sort of wire routing that asks for a configuration of wires connecting a given set of pins such that the wires do not go through obstacles, and that the total wire length be minimum (motivated by delay minimization). We view the given chip as a rectangular grid, and study this problem by means of a graph problem, Rectilinear Steiner Tree (RST) construction, which asks for a connected graph that is composed of horizontal or vertical line segments and that spans a given set of points such that the total length of its edges is minimum. The decision problem version of RST construction (i.e., whether there is a rectilinear tree of size less than or equal to k, that connects the given points) is NP-hard [1].

3.1 Problem description

Consider an undirected graph (V, E) with a positive number, called the *length*, assigned to every edge. A *Steiner tree* for a set S of vertices is a tree (V', E') that is a subgraph of (V, E) where $S \subseteq V'$ and the total length of edges in E' is minimum. In the *rectilinear Steiner tree* problem (**RST**), the goal is to find a Steiner tree in the case when the edges of the given graph (V, E) are horizontal and vertical line segments in a plane. We assume that the graph is specified as a subset of the grid of unit squares, so that the goal is to find a total number of unit segments covered by the tree is minimum. Since **RST** is too hard for OPL and CLASP, we consider a decision problem that corresponds to it:

RST-DEC Given a subgraph G of a rectangular grid, a source point s, a set S of sink points, and an integer k, decide that there is a subgraph of G such that
R1 every point in the subgraph is included in a path connecting a sink to the source,
R2 each sink i and the source are included in the same path (Path i) in the subgraph,
R3 in Path i, each end point is not connected to more than one point,
R4 in Path i, each internal point is connected to exactly two points, and
R5 the total length of the subgraph is less than or equal to k.

Conditions R1–R4 ensure that the subgraph connects every sink point to the source. For a sufficiently small lower bound k, a solution to **RST-DEC** is a solution to **RST**.

3.2 Representation of RST-DEC in the language of LPARSE

According to the formulation of [13], every path connecting a sink point to the source point is characterized by the truth values of the atoms `in(h,N,X,Y)` ("the horizontal segment connecting the points `(X,Y)` and `(X+1,Y)` occurs in Path `N`") and `in(v,N,X,Y)` ("the vertical segment connecting the points `(X,Y)` and `(X,Y+1)` occurs in Path `N`").

We first "generate" sets of atoms of the form `in(S,N,X,Y)` by the rule

```
{in(S,N,X,Y)} :- segment(S), path(N), point(X,Y).
```

where `point(X,Y)`, defined in terms of `obstacle(X1,X2,Y)`, expresses that the grid point `(X,Y)` is not blocked by an obstacle. Such a subgraph already satisfies R1.

After eliminating the sets that contain segments connecting a point on the grid that is not blocked by an obstacle to the point that is blocked by an obstacle (or is out of the grid), the remaining sets are "tested" with some constraints expressing R2–R5. To express R2, i.e., the two points specified in the problem description file as the end points of Path `N` belong to that path, we define the atom `at(N,X,Y)` ("the point `(X,Y)` is in Path `N`") and include the constraint

```
:- not at(N,X,Y), ends(N,X,Y).
```

We need to make sure, furthermore, R3, i.e., each of these two points is connected to only one other point in the path, whereas R4, i.e., every other point of Path `N` is connected to exactly two points. For that, we define the atom `at(N,X,Y,D)` ("the unit segment that begins at the point `(X,Y)` and goes in the direction `D` occurs in Path `N`"). We ensure R3 by the constraint

```
:- 2{at(N,X,Y,D):direct(D)}, path(N), ends(N,X,Y).
```

and R4 by the constraints

```
:- 1{at(N,X,Y,D):direct(D)}1, path(N), point(X,Y), not ends(N,X,Y).
:- 3{at(N,X,Y,D):direct(D)}, path(N), point(X,Y), not ends(N,X,Y).
```

Finally, for R5, we eliminate the sets where the total wire length is larger than `k` by adding the constraint

```
:- k+1 {covered(S,X,Y):segment(S):point(X,Y)}.
```

where `covered(S,X,Y)` describes the unit segments covered by the generated graph.

3.3 Representation of RST-DEC in the language of OPL

Every path connecting a sink point to the source point is characterized by the truth values of the variables `inGraph[S,<k1,k2>,<X,Y>]`, similar to atoms `in(S,N,X,Y)` of the ASP formulation above. Here instead of the label `N` of a sink point, its coordinates are used; `in` is not used as the name of the variable since it is a reserved word. Possible values of these variables are "generated" by the following declaration:

```
dvar int inGraph[segment,sink,all_grid] in 0..1;
```

where `segment` describes the set of segments "v" and "h", `sink` describes the set of sink points, and `all_grid` describes the set of all grid points. The subgraph described by values of these variables already satisfies R1.

After obtaining the set `V` of all grid points that are not covered by obstacles, and eliminate the subgraphs that contain segments connecting a point in `V` to a point that is not in `V`, Conditions R2 and R3 are described by a set of constraints like

```
forall(<k1,k2> in sink, <X,Y> in V:
 ((X==source.x && Y==source.y) || (X==k1 && Y==k2 )) &&
  <X,Y-1> in V && <X-1,Y> in V && <X+1,Y> in V && <X,Y+1> in V)
   inGraph["v",<k1,k2>,<X,Y>]+inGraph["h",<k1,k2>,<X,Y>]+
   inGraph["v",<k1,k2>,<X,Y-1>]+inGraph["h",<k1,k2>,<X-1,Y>]==1;
```

Here the third line is required to declare the domain of each pair (e.g., `<X,Y-1>` is in `V`), mentioned in the constraint. Therefore we have 14 other constraints for other possible declarations of the domains of these pairs.

To describe R4, as in the ASP formulation, first we introduce an auxiliary variable of the form `at[<k1,k2>,<X,Y>]`. Then we add a set of constraints, including

```
forall(<k1,k2> in sink, <X,Y> in V:
 ((X!=source.x || Y!=source.y) && (X!=k1 || Y!=k2)) &&
  <X,Y-1> in V && <X-1,Y> in V && <X,Y+1> in V && <X+1,Y> in V)
   inGraph["v",<k1,k2>,<X,Y>]+inGraph["h",<k1,k2>,<X,Y>]+
   inGraph["v",<k1,k2>,<X,Y-1>]+inGraph["h",<k1,k2>,<X-1,Y>]
   ==at[<k1,k2>,<X,Y>]*2;
```

Like in the description of R2 and R3, there are 14 other constraints for other possible declarations of the domains of the pairs that appear in the constraint.

To describe R5, as in the ASP formulation, first we introduce an auxiliary variable of the form `covered[<X,Y>]` (“`Point (X,Y)` is covered by the generated subgraph”). Then we add the constraint:

```
sum(<X,Y> in all_grid) covered[<X,Y>] <= k+1;
```

3.4 A comparison of the formulations

We compare the formulations above both from the point of view of representation (in terms of expressivity, and elaboration tolerance), and from the point of view of computational efficiency (in terms of CPU time, and program size). Then we discuss possible ways of improving the computational efficiency taking into account other formulations of the problems based on different mathematical models.

Elaboration tolerance Consider a variation of wire routing that requires some restrictions on lengths of the wires connecting the sink pins to the source pin, that is to say, on signal delays. We can express in ASP that any wire cannot be longer than a specific value, say `l`, by adding to the problem description the constraint

```
:- l+2 {at(N,X,Y):point(X,Y)}, path(N).
```

(no path connecting the source pin to a sink pin is allowed to cover `l+1` unit segments on the grid). This elaboration can be easily tolerated by CP/ILP representation as well, by adding to the problem description the constraint

```
forall(<k1,k2> in sink)
  sum(<X,Y> in V) at[<k1,k2>,<X,Y>] <= l+1;
```

In the formulations of **RST-DEC** above if the upper bound k on the total number of the segments covered by the paths is not small enough then the graph generated by any of these programs may not be a tree, i.e., there may be cycles in it. A path from the source to a sink generated by the program cannot contain cycles, but the graph can contain cycles that are completely disjoint from these paths, as well as cycles consisting of parts of several different paths. So let us consider a variation of **RST-DEC** where the generated subgraph is a tree. Such a modification can be tolerated by ASP by adding to the program the constraint

```
:- at(N,X,Y), at(N1,X,Y), not reachable(N,N1,X,Y),
   point(X,Y), path(N;N1), N<=N1.
```

where `reachable(N,N1,X,Y)` expresses that point `(X,Y)` is reachable from the source point via the common segments of the paths `N` and `N1` (`N`$\leq$`N1`). This predicate has a straightforward recursive definition. The base case is defined by the rule

```
reachable(N,N1,X,Y) :- source(X,Y), path(N;N1), N <= N1.
```

The inductive step is defined by the rule

```
reachable(N,N1,X+1,Y) :- reachable(N,N1,X,Y), path(N;N1), N <= N1,
  in(h,N,X,Y), in(h,N1,X,Y), point(X,Y), point(X+1,Y).
```

and a similar rule for vertical segments. In the case of N=N1, the constraint above expresses that every point covered by a path is reachable from the source point. This guarantees that the graph is connected and thus eliminates cycles of the first kind. In the case of N≠N1, the constraint expresses that the intersection of two paths is contiguous and thus eliminates cycles of the second kind. With this formulation, one can generate a rectilinear tree of length at most k, using an answer set solver. On the other hand, this elaboration cannot be tolerated in CP/ILP formulations easily (without enumerating all possible paths, and/or introducing too many auxiliary variables).

Heuristics As in [13], we considered two heuristics. First, we force each path connecting the source to a sink i to be covered by two circles of a given radius, one around the source and the other around sink i, so that other parts of the grid are not searched. Second, we force that no two unit segments are adjacent. For instance, the second heuristic is expressed in the language of OPL by the constraint

```
forall(<X,Y> in V,<k1,k2> in sink, <k3,k4> in sink, <X,Y+1> in V)
   inGraph["h",<k1,k2>,<X,Y>]+inGraph["h",<k3,k4>,<X,Y+1>]<=1;
```

and in the language of LPARSE by the constraint

```
:- in(h,X,Y), in(h,X,Y+1), point(X,Y), point(X,Y+1).
```

Other formulations Another mathematical model of **RST-DEC**, introduced in particular for an ILP formulation, is flow-based, like in [14]. The idea is to introduce a dummy node, say D, and assign some flow (of value 0 or 1) to every edge on a directed Path i that connects D to an end point i, so that the edges that have positive flow relative to Path i are included in the subgraph. Then the subgraph is a solution to **RST-DEC** if the following hold: the total outflow of the source (resp. sink) point relative to Path i is one more (resp. less) than its total inflow; the total outflow of every internal point relative to Path i is equal to its total inflow. Representations of this mathematical model are available at [6].

Computational efficiency To make the computation more efficient we added to each formulation some heuristics mentioned above. Then, with each formulation, we tried to solve four problem instances of [13], using the answer set solver CMODELS (Version 3.74) with LPARSE (Version 1.0.17) and MINISAT (Version 2.0), and OPL (Version 5.5) on a machine with Xeon 1.5GHz CPU with 4x512MB RAM running RedHat Linux (Version 4.3). For each problem instance of **RST**, we considered two instances of **RST-DEC**: one with the optimal value of k (where the answer to the decision problem is "yes"—to generate a solution), and the other with 1 less than the optimal (where the answer to the decision problem is "no"—to verify the minimality of the solution). As in [13], when generating a solution we added to the programs some rules/constraints describing some heuristics; verification of minimality does not involve any heuristics.

Table 1 compares the computation time (CPU time in sec.s), and Table 2 compares the program size (number of variables and constraints in the case of OPL, and number of atoms and clauses in the case of CMODELS) of formulations of **RST-DEC** described

Table 1. Experimental results for **RST-DEC**: computation times

Problem	# of sink pins	k	CPU time (sec.)	
			ILP – OPL	ASP – CMODELS
A	15	58	4.64	1.38
		57	-	30.82
B	20	68	10.35	2.30
		67	-	416.40
C	25	74	149.94	2.75
		73	-	-
D	30	76	265.73	2.76
		75	-	-

Table 2. Experimental results for **RST-DEC**: program size

Problem	# of sinks	k	ILP – OPL		ASP – CMODELS	
			# of variables	# of constraints	# of atoms	# of clauses
A	15	58	11776	72213	36643	110381
		57	11776	27477	34815	104848
B	20	68	15616	115429	41989	126943
		67	15616	36597	42483	128238
C	25	74	19456	168277	46200	140060
		73	19456	45717	49455	149537
D	30	76	23296	230699	50248	152784
		75	23296	54837	56013	169592

above. In Table 1, a dash - indicates that the problem could not be solved in 900 sec.s. For instance, consider Problem A. An RST is computed using OPL, when $k = 58$, in 4.64 sec.s with the ILP formulation of **RST-DEC** with some heuristics. With the ILP formulation of **RST-DEC** only, OPL could not verify in less than 900 sec.s that there is no tree of size $k = 57$. In the former (resp. latter) computation, the program contains 11776 variables and 72213 (resp. 27477) constraints. Using CMODELS, a solution of size 58 is generated in 1.38 sec.s, and its minimality is verified in 30.82 sec.s. With the CP formulation, OPL could not generate a solution for any of the problems in 900 sec.s.

As observed in [13], when we add to the formulations constraints about length restrictions, the computation time increases. For instance, using CMODELS, for Problem A, a tree of size 58 with $l = 16$ is generated in 2.87 sec.s. When we consider the computation of a tree of some length less than or equal to k (not necessarily the smallest one) then the computation time CMODELS does not increase as much as the program size does. For instance, for Problem D (with $k = 100$), CMODELS computes a tree of size 88 in 3.62 sec.s with a program with 157163 atoms and 603716 clauses.

The flow-based ILP formulation significantly improves computational efficiency: for Problem C with $k = 74$, the CPU time decreases to 3.17 sec.s, and the number of constraints to 10241. The ASP formulation does not improve the computational efficiency: for Problem C, the CPU time increases to 62.70 sec.s, and the number of clauses to 560520. No problems can be solved in 900 sec.s with the CP formulation.

4 Haplotype Inference Pure Parsimony Problem

A genotype is the specific genetic makeup of an individual. Each genotype has two copies, one from the mother and one from the father. These two copies are called haplotypes, and they combine to form the genotype. Due to technological limitations, we have access to genotype data rather than haplotype data. However, most of the genetic information is contained in haplotypes, which can be used for early diagnosis of diseases, detection of transplant rejection and creation of evolutionary trees. Motivated by the goal of identifying maternal and paternal inheritance to be able to map disease genes, and find the set of genes responsible for a particular disease, researchers have been studying how to infer haplotypes that form a given set of genotypes.

Different pairs of haplotypes may form the same genotype, and this ambiguity makes it difficult to find the "correct" haplotypes that explain the given genotype. For that, researchers have studied a slight modification of the haplotype inference problem, *Haplotype Inference with Pure Parsimony* (**HIPP**) [2]—to infer a minimal set of haplotypes that explain the given genotypes. The decision version of **HIPP** (i.e., deciding that a set of k haplotypes that explain the given genotypes exists) is NP-hard [2, 3].

4.1 Problem description

A standard definition of the concept of two haplotypes "explaining" a genotype appears in [2]. According to this definition, we view a genotype as a vector of sites, each site having a value 0, 1, or 2; and a haplotype as a vector of sites, each site having a value 0 or 1. A site of a genotype is *ambiguous* if its value is 2; and *resolved* otherwise. Two haplotypes h_1 and h_2 *form (explain)* a genotype g if for every site j the following hold:

- if $g[j] = 2$ then $h_1[j] = 0$ and $h_2[j] = 1$ or $h_1[j] = 1$ and $h_2[j] = 0$;
- if $g[j] = 1$ then $h_1[j] = 1$ and $h_2[j] = 1$; and
- if $g[j] = 0$ then $h_1[j] = 0$ and $h_2[j] = 0$.

For instance, the genotype 20110 can be explained by the haplotypes 10110 and 00110.

We consider the following decision problem version of **HIPP**:

HIPP-DEC Given a set G of n genotypes each with m sites, and a positive integer k, decide whether there is a set H of k haplotypes such that each genotype in G is explained by two haplotypes in H.

For a sufficiently small k, a solution to **HIPP-DEC** is a solution to **HIPP** as well.

For this problem, H is a solution if the following hold:

C1 Every genotype g in G is mapped to two haplotypes in H.
C2 For every genotype g in G, for every ambiguous site j of g, the values of the j'th sites of these haplotypes are different.
C3 For every genotype g in G, for every resolved site j of g, the values of the j'th site of these haplotypes are $g[j]$.

4.2 Representations of HIPP-DEC

In the language of OPL, the given set of genotypes are declared as a matrix of size nm, and the set of k haplotypes to be inferred is declared as a matrix of size km. CP and

```
using CP;
int n = ...;   // n genotypes
int m = ...;   // m sites
int k = ...;   // k haplotypes
int g[1..n,1..m] = ...;         // g[i,j]=0,1,2

// Generate a matrix of k haplotypes
dvar int h[1..k,1..m] in 0..1;  // h[i,j]=0,1

// Test wrt the given constraints C1--C3
// C1 Map every genotype i to two haplotypes s[1][i] and s2[2][i]
dvar int s[1..2][1..n] in 1..k;
subject to {
   // C2 & C3
   forall(i in 1..n, j in 1..m)
      if(g[i,j] == 2){h[s[1,i],j] != h[s[2,i],j];}
      else{
        h[s[1,i],j] == h[s[2,i],j] && h[s[2,i],j] == g[i,j];}  }
```

Fig. 1. A CP representation of **HIPP-DEC** in the language of OPL.

```
int n = ...;   // n genotypes
int m = ...;   // m sites
int k = ...;   // k haplotypes
int g[1..n,1..m] = ...;   // g[i,j]=0,1,2

// Generate a matrix of k haplotypes
dvar int h[1..k,1..m] in 0..1;
// and a matrix for mapping two haplotypes to one genotype
dvar int s[1..k,1..k,1..n] in 0..1;

// Test wrt the given constraints C1--C3
subject to {
  // C1
  forall(i in 1..n) sum(i1 in 1..k,i2 in 1..k) s[i1,i2,i]==1;
  // C2 & C3
  forall(i in 1..n,i1 in 1..k, i2 in 1..k)
    sum(j in 1..m) ((h[i1,j] == h[i2,j] && h[i2,j] == g[i,j]) ||
        (h[i1,j] != h[i2,j] && g[i,j]==2)) >= m*s[i1,i2,i];}
```

Fig. 2. An ILP representation of **HIPP-DEC** in the language of OPL.

ILP formulations of **HIPP-DEC** in the language of OPL are shown in Fig.s 1 and 2. In the CP representation, we use functions $s[1, i]$ and $s[2, i]$ to describe the first and second haplotype explaining Genotype i. In the ILP representation, array indices cannot be decision variables, so we introduce the variables $s[i_1, i_2, i]$ to describe the explanation of Genotype i by haplotypes i_1 and i_2. In the ASP representation, since functions are not allowed, we describe the value of the j'th site of a genotype g by atoms of the form $amb(g, j)$. Consider some answer set X describing a solution. Then

– $amb(g, j) \in X$ iff $g[j] = 2$,

```
geno(1..n).        % n genotypes
site(1..m).        % m sites
haplo(1..k).       % k haplotypes
index(1..2).       % two haplotypes explaining a genotype
#domain geno(G), index(I), site(J).  % sorts of variables

% Generate a set of k haplotypes
{h(H,J)} :- haplo(H).

% Test wrt the given constraints C1--C3
% C1
1{s(I,G,H):haplo(H)}1.
% C2
:- s(1,G,H1), s(2,G,H2), amb(G,J), h(H1,J), h(H2,J), haplo(H1;H2).
:- s(1,G,H1), s(2,G,H2), amb(G,J), not h(H1,J), not h(H2,J), haplo(H1;H2).
% C3
:- not h(H,J), s(I,G,H), -amb(G,J), haplo(H).
:- h(H,J), s(I,G,H), not -amb(G,J), not amb(G,J), haplo(H).
```

Fig. 3. A representation of **HIPP-DEC** in the language of LPARSE.

- $\neg amb(g, j) \in X$ iff $g[j] = 1$, and
- $amb(g, j), \neg amb(g, j) \notin X$ iff $g[j] = 0$.

Similarly, we describe the value of the j'th site of a haplotype i by atoms of the form $h(i, j)$. Consider some answer set X describing a solution. Then $h(i, j) \in X$ iff $i[j] = 1$, and $h(i, j) \notin X$ iff $i[j] = 0$. Then the formulation of **HIPP-DEC** in the language of LPARSE follows from the description in Section 4.1 as in Fig. 3.

4.3 A comparison of the formulations

As in wire routing problems, we compare the formulations above both from the point of view of representation, and from the point of view of computational efficiency.

Representing **HIPP** Now we are given various formulations for **HIPP-DEC**, we can find a solution to **HIPP** by iteratively solving **HIPP-DEC** to find the minimum k (e.g., by doing binary search). On the other hand, all three formalisms have special constructs to solve optimization problems. Let us investigate how the representations in the previous section can be minimally modified to obtain formulations for **HIPP**.

In the CP formulation in Fig. 1, we change the size of the matrix describing haplotypes to $2nm$, and the mapping of genotypes to haplotypes accordingly, and add before the constraints the following minimization statement:

```
minimize max(i in 1..2,j in 1..n) s[i][j];
```

However, in the ILP formulation in Fig. 2, we need to ensure an increasing order of indices for two haplotypes explaining a genotype, by modifying the first constraint:

```
sum(i1 in 1..2*n,i2 in 1..2*n) ((i1 <= i2)*s[i1,i2,i]) == 1;
```

Then, we add the minimization statement:

```
minimize max(i1 in 1..2*n,i2 in 1..2*n,i in 1..n) i2*s[i1,i2,i];
```

As for the ASP encoding, first we modify the range of haplotypes to $1..2n$ and add the minimization statement to the end of the program

```
minimize [mapped(H):haplo(H)].
```

where `mapped` describes the haplotypes that explain some genotype.

Symmetry breaking constraints and adding auxiliary atoms/variables Two haplotypes are different if they have a site with different values; otherwise they are identical. To improve computational efficiency, we add to the formulations of **HIPP-DEC** the following constraint to ensure that the haplotypes in the inferred set H of haplotypes are unique: (S1) Every haplotype h in H is different from all other haplotypes in H. They can be expressed in the language of OPL as follows:

```
forall(i1 in 1..k,i2 in 1..k)
  d[i1,i2] <= sum(j in 1..m) (abs(h[i1][j] - h[i2][j]));
```

where `d[i,j]=1` iff Haplotypes `i`, `j` are different, and in the language of LPARSE by

```
:- {different(H1,H2,J):site(J)}0, haplo(H1;H2), H1<H2.
```

where `different(H1,H2,J)` describes that two haplotypes `H1` and `H2` have different values at site `J`.

Instead of the constraints S1, let us consider a different formulation which eliminates some of the symmetries only: the idea is (S2) not to generate a haplotype i unless haplotype $i-1$ is already generated.[6] For instance, in the ASP formulation, we add

```
{generate(1)}.
{generate(H)} :- haplo(H), generate(H-1).
:- s(I,G,H), haplo(H), not generate(H), geno(G), index(I).
```

and modify the first choice rule as follows:

```
{h(H,J)} :- haplo(H), generate(H), site(J).
```

In the case of CP and ILP, we add to the program

```
forall(i in 1..k) (-i+2) + sum(j in 1..i-1) d[i,j] == 1;
```

Elaboration tolerance To transform representations of **HIPP-DEC** to those **HIPP**, we modify some parameters of predicates; for symmetry breaking (e.g., with S1) we simply add some constraints or rules to the formulations. In that sense our formalizations are more tolerant to the latter sort of modifications.

Other formulations Another definition of **HIPP** (and thus **HIPP-DEC**), especially tuned for ILP, is due to Brown and Harrower [15]. According to that definition, a genotype is *ambiguous* if its value is 1; and *resolved* otherwise; and two haplotypes h_1 and h_2 *form (or explain)* a genotype g if for every site j the following hold: $g[j] = h_1[j] + h_2[j]$. Then, a set H of k haplotypes is a solution to **HIPP-DEC** if, for every genotype g in G, there exist two haplotypes h_1 and h_2 in H such that, for every site j, $g[j] = h_1[j] + h_2[j]$. This constraint can be represented in CP by

```
forall(i in 1..n, j in 1..m) h[s[1,i],j] + h[s[2,i],j] == g[i,j];
```

in ILP by

```
forall(i in 1..n, i1 in 1..k, i2 in 1..k)
   sum(j in 1..m) ((h[i1,j]+h[i2,j]==g[i,j])) >= m*s[i1,i2,i];
```

Table 3. Experimental results for **HIPP-DEC**: computation times

Problem	n	m	k	CPU time (sec.)	
				CP (Fig. 1) – OPL	ASP (Fig. 3) – CLASP
P0	5	4	5	0.01	0.01
			4	0.03	0.01
P1	5	10	8	0.14	0.06
			7	390.6	0.27
P2	8	9	8	0.19	0.07
			7	98.85	0.19
P3	13	12	12	1.74	0.42
			11	-	-

Table 4. Experimental results **HIPP-DEC**: program size

Problem	n	m	k	CP (Fig. 1) – OPL		ASP (Fig. 3) – CLASP	
				# of variables	# of constraints	# of variables	# of rules
P0	5	4	5	220	69	531	535
			4	216	69	381	394
P1	5	10	8	555	167	2885	2880
			7	545	167	2289	2293
P2	8	9	8	800	257	3177	3151
			7	791	257	2560	2549
P3	13	12	12	1676	541	15341	15199
			11	1664	541	13143	13026

and in ASP by

```
:- s(G,H1,H2), h(H1,J,N1), h(H2,J,N2), not g(G,J,N1+N2), haplo(H1;H2).
```

Here `s(G,H1,H2)` describes Haplotypes `H1,H2` explaining Genotype `G`, and `h(H,J,N)` describes the value `N` of the `J`th site of Haplotype `N`.

In another approach, we can get rid of large matrices describing which haplotypes form which genotypes; and enforce that each genotype i be explained by haplotypes $2i$ and $2i-1$. Its formulations can be found at the web page [6].

Computational efficiency With the formulations above, we tried to solve four problem instances of **HIPP**, randomly chosen amongst the ones tested in [15], using the answer set solver CLASP (Version 1.0.4) and ILOG OPL (Version 5.5) with CP-OPTIMIZER (Version 1.1), on a machine with Intel Centrino 1.8GHz CPU and 1 GB of RAM running on Windows XP. For each instance of **HIPP**, we considered two instances of **HIPP-DEC**: one with the optimal value of k, and the other with 1 less than the optimal.

Table 3 compares the computation time (CPU time in sec.s), and Table 4 compares the program size (number of variables and constraints in the case of OPL, and number of variables and rules in the case of CLASP) of formulations of Fig.s 1–3, for the four **HIPP** instances. In Table 3, a dash - indicates that the problem could not be solved in 900 sec.s. For instance, consider the problem P0. A set of 5 haplotypes that explain the given genotypes is computed in 0.01 sec.s using OPL with the CP formulation;

with the same formulation and the solver, it is verified in 0.03 sec.s that there is no set of 4 haplotypes that explain the given genotypes. In the computation of the former (resp. latter) instance, the theory contains 555 (resp. 545) variables and 167 (resp. 167) constraints. On the other hand, neither solver terminates with a decision for problem P4 with $k = 11$. With the ILP formulation of **HIPP-DEC**, for P0, OPL finds a solution (with $k = 8$) in 8.85 sec.s where the theory contains 2656 variables and 405 constraints, and verifies its minimality (with $k = 7$) in 34.20 sec.s where the theory contains 4145 variables and 630 constraints; for other problems OPL could not generate a solution in 900 sec.s (this is why we show in the tables only results of our experiments with the CP and ASP formulations). ILP formulations of problems have larger program sizes, mainly due to the larger size of `s` (since indices cannot be decision variables as in CP).

Adding to these formulations symmetry breaking constraints does not improve the computational efficiency. For instance, for P2 with $k = 8$, with the CP formulation, the computation time increases from 0.19 sec.s to 0.29 sec.s with the constraints S1, and to 0.23 sec.s with the constraints S2. On the other hand, adding S2 to the formulations of **HIPP** (obtained from those of **HIPP-DEC** as described above) decreases the computation time: while P2 can not be solved with the CP formulation in 900 sec.s, it can be solved in 389.68 sec.s when these constraints are added; similarly, the computation time reduces from 557.42 sec.s to 0.68 sec.s with the ASP formulation.

With the alternative definition of **HIPP-DEC** due to [15], with the CP formulation, the computation time and the number of constraints decrease (at least by factor of 2). With the ASP formulation, both the computation time and the program size increase. With the ILP formulation, OPL performs much better (the CPU time and the number of constraints decrease at least by a factor of 8), but not as good as with the CP or ASP formulations. Adding symmetry breaking constraints to these formulations of **HIPP-DEC** does not improve the computational efficiency. On the other hand, when added to the respective CP and ASP formulations of **HIPP** the CPU time decreases: S1 (resp. S2) reduces the CPU time for P2 by a factor of at least 3 (resp. 6).

5 Conclusion

We studied two challenging problems, wire routing and haplotype inference, in the context of three declarative programming paradigms, ASP, CP, ILP, used usually by different research communities in AI/CP/OR. We represented these two problems in each formalism systematically in the same way, and, also due to the declarative nature of the formalisms, we observed that they are tolerant to almost all elaborations (e.g., adding symmetry breaking, transforming **HIPP-DEC** to **HIPP**) in the sense of McCarthy [4]. We observed some differences in the representations due to specifics of the input languages of the solvers. For instance, in CP formulations, OPL allows us to declare indices of a matrix as decision variables, and, as observed in our experiments with **HIPP-DEC** this leads to less number of atoms compared to the ILP formulations. For instance, in all formalisms one can express constraints on the cardinality of a set, as seen in the formulation of **RST-DEC** with length restrictions; in CP and ILP one can use the aggregate `sum`, and in ASP one can use a cardinality expression.

Some differences are due to the expressivity of these paradigms. In ASP formulations, one can include recursive definitions, like the definition of reachability in the formulation of **RST-DEC**; in CP and ILP formulations, we have to enumerate all possibilities (cf. formulations of Steiner tree problem in [14]), or introduce too many auxiliary variables. For instance, to express the uniqueness of haplotypes generated so far, in CP and ILP $2k^2$ auxiliary atoms (of the forms `d[i,j]==0` and `d[i,j]==1`) are introduced. In ASP, a recursive definition of k atoms (of the form `generate(i)`) suffices. In the formalization of **HIPP-DEC**, due to the presence of negation as failure in ASP, it is sufficient to introduce nm atoms to describe the given genotypes; in CP and ILP formulations, genotypes are described by $3nm$ atoms. On the other hand, being able to represent functions and matrices in CP and ILP helps us formulate some problems more concisely, as observed in the formalization of **HIPP-DEC** relative to [15].

Adding domain specific heuristics (e.g., the ones for **RST-DEC**) improves computational efficiency; on the other hand, adding domain independent constraints (e.g., symmetry breaking in **HIPP-DEC**) not always improves the computational efficiency.

Besides comparing ASP, CP, ILP on wire routing and haplotype inference problems, our paper introduces new approaches/formulations (e.g., CP/ILP formulations of **RST-DEC**, ASP formulation of **HIPP-DEC**) to solve these problems, sometimes inspired by formulations in other formalisms. All representations are available at [6]. A comparison of these approaches with the existing ones is a part of future work.

References

1. Garey, M.R., Johnson, D.S.: The rectilinear Steiner tree problem is NP complete. SIAM Journal of Applied Mathematics **32** (1977) 826–834
2. Gusfield, D.: Haplotype inference by pure parsimony. In: Proc. of CPM. (2003) 144–155
3. Lancia, G., Pinotti, M.C., Rizzi, R.: Haplotyping populations by pure parsimony: Complexity of exact and approximation algorithms. INFORMS Jour. on Computing **16** (2004) 348–359
4. McCarthy, J.: Elaboration tolerance. In: Proc. of CommonSense. (1998)
5. Cadoli, M., Mancini, T., Micaletto, D., Patrizi, F.: Evaluating asp and commercial solvers on the csplib. In: Proc. of ECAI. (2006) 68–72
6. `http://people.sabanciuniv.edu/~esraerdem/benchmarks/asp-cp-ilp.html`
7. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
8. Hoffman, K.L., Padberg, M.: Encyclopedia of Operations Research and Management Science. Kluwer, Massachusetts (2001)
9. Rossi, F., van Beek, P., Walsh, T.: Handbook of Constraint Programming. Elsevier (2006)
10. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Logic Programming: Proc. of the 5th International Conference and Symposium. (1988) 1070–1080
11. Ferraris, P., Lifschitz, V.: Mathematical foundations of answer set programming. In: We Will Show Them! Essays in Honour of Dov Gabbay. Volume 1. (2005) 615–664
12. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138** (2002) 181–234
13. Erdem, E., Wong, M.D.F.: Rectilinear Steiner Tree construction using answer set programming. In: Proc. of ICLP. (2004) 386–399
14. Maculan, N., Plateau, G., Lisser, A.: Integer linear models with a polynomial number of variables and constraints for some classical combinatorial problems. Pesquisa Operacional **23**(1) (2003) 161–168
15. Brown, D., Harrower, I.: Integer programming approaches to haplotype inference by pure parsimony. IEEE/ACM Trans. on Bioinformatics and Comp. Biology **3** (2006) 348–359