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

Elvin Çoban[1], Esra Erdem[2], and Ferhan Türe[3]

[1] Operations Management Tepper School of Business, Carnegie Mellon University, USA
[2] Faculty of Engineering and Natural Sciences, Sabancı University, Turkey
[3] Department of Computer Science, University of Maryland, USA

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 problems in each formalism in a systematic way, compare the 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.

1 Introduction

Our goal is to compare the three declarative programming paradigms, Answer Set Programming (ASP), Constraint Programming (CP), and Integer Linear Programming (ILP), on some challenging applications, both from the point of view of knowledge representation (KR) and from the point of view of computational efficiency. This paper describes results of some preliminary work in this direction.

Computational problems In this paper, we study two challenging applications, wire routing and haplotype inference.

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, haplotype inference asks for a set of haplotypes that form a given set of genotypes. We consider in particular a slight modification of this problem that asks for a minimal set of haplotypes that explain the given genotypes; the decision version of this problem is NP-hard [2, 3].

Representation of problems We consider the decision problems corresponding to the problems above. For each computational problem, we decide for a straightforward mathematical model to solve it. Then we represent each decision problem in ASP, CP, and ILP, with respect to the selected mathematical model, 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. The idea is to define concepts in each language in a similar way.

Comparing ASP, CP, and ILP from the point of view of KR We compare representations of a problem with respect to the following criterion:

- One of the most important problems in KR is elaboration tolerance of representations, pointed out by John McCarthy in [4]. A representation 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. Therefore, one criteria to compare the representations of problems is elaboration tolerance: we check how tolerant the representations are to elaborations or exceptions. In other words, we investigate how much we need to modify a representation of a computational problem, like wire routing, to solve a variation of that problem, like wire routing where restrictions are put on the lengths of wires.
- We investigate how easy it is to express, in each paradigm, some concepts and constraints, like reachability and cardinality constraints, that are required for straightforward representations of problems.

Comparing ASP, CP, and ILP from the point of view of computational efficiency We compare formulations of problems from the point of view of computational efficiency, in terms of CPU time and program size. In our experiments, we use the ASP solvers CLASP[4] and CMODELS;[5] and the commercial CP and ILP solver ILOG OPL.[6]

Other comparisons In addition to comparing the straightforward representations of the problems as described above, we consider various reformulations of these problems:

- We investigate possible ways of improving the straightforward representations, from the point of view of computational efficiency (e.g., by adding symmetry breaking constraints, introducing auxiliary variables, and adding redundant constraints) taking into account elaboration tolerance.
- We consider other mathematical models to solve the computational problems, and represent them in each paradigm in a systematic way as described above. In particular, we consider alternative mathematical models "tuned" for one particular formalism, and examine the corresponding reformulations in other formalisms.

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

[4] `http://www.cs.uni-potsdam.de/clasp`

[5] `http://www.cs.utexas.edu/users/tag/cmodels`

[6] `http://www.ilog.com/products/solver`

a continuation and/or a complement of this study. On the other hand, our paper not only reports a comparison of ASP, CP, and ILP based on our experiences, but also introduces new approaches and formulations to solving wire routing and haplotype inference.

We refer the readers to [6–8] for more information about ASP, ILP, CP. All formulations, and further experimental results are available at [9].

2 Wire Routing

We study wire routing by means of a graph problem, Rectilinear Steiner Tree (RST) construction, defined as follows. 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 **RST** problem 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; the length of an edge is the total number of unit segments covered by the edge. We consider the following decision problem that corresponds to **RST**:

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 T of G such that
- R1 every point in T 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 T,
- 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 T 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**.

Another mathematical model for **RST**, introduced in particular for an ILP formulation, is flow-based. 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 Path i that connects D to an end point i; the edges that have positive flow are included in the subgraph.

2.1 Formulations of wire routing

The former mathematical model for **RST-DEC** (i.e., the tree-based approach) described in the previous section has already been formulated in ASP [13]. The latter model (i.e., the flow-based approach) has been formulated in ILP [14]. We have represented the former model in ILP and CP; and the latter model in ASP and CP. In the following we present the formulations of **RST-DEC** with respect to the tree-based approach, in the language of LPARSE and in the language of OPL. All formulations are available at [9].

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 `(X,Y)` and `(X+1,Y)` occurs in Path `N`") and `in(v,N,X,Y)` ("the vertical segment connecting `(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 rules

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

where `point(X,Y)` expresses that the grid point `(X,Y)` is not blocked by an obstacle. Such a subgraph satisfies R1.

After eliminating the sets that contain segments connecting a point on the grid that is not blocked by an obstacle to a 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 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 constraints

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

We need to make sure, furthermore, R3, i.e., each of these end points is connected to only one other point in Path `N`, and R4, i.e., every other point of Path `N` is connected to exactly two points. For that, we define the auxiliary 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 constraints

```
:- 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 constraints

```
:- 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.

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={"v","h"}`, `sink` is the set of sink points, and `all_grid` is 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 eliminating the subgraphs that contain segments connecting a point in `V` to a point that is not in `V`, Conditions R2, 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 fourth and the fifth lines are 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, we introduce an auxiliary variable of the form `at[<k1,k2>,<X,Y>]`, and 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;
```

2.2 A comparison of the formulations

We compare the formulations discussed 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, program size).

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;
```

The formulations of **RST-DEC** with respect to R1–R5 do not include an explicit definition of a path; connectedness of a tree is guaranteed when we find the shortest path. However, 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. 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.

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 the CP and ILP formulations easily (without enumerating all possible paths, and/or introducing too many auxiliary variables). In this sense, ASP is more elaboration tolerant.

Computational efficiency With each formulation of **RST-DEC**, we solved the four problem instances, of [13], using CMODELS 3.74 with LPARSE 1.0.17 and MINISAT 2.0, and ILOG OPL 5.5 on a machine with Xeon 1.5GHz CPU with 4x512MB RAM running RedHat Linux 4.3. For each problem instance of **RST**, we present results for two instances of **RST-DEC**: one with the optimal value of k (to generate a solution), and the other with 1 less than the optimal (to verify the minimality of the solution). As in [13], the optimal value of k was computed iteratively. When generating a solution we added to the programs some rules/constraints describing some heuristics (e.g., forcing 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); verification of minimality does not involve those 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 above. In Table 1, a dash - indicates that the problem could not be solved in 900 sec.s.

Table 1. Experimental results for **RST-DEC**: computation times (CPU sec.s)

Problem	# of sink pins	k	Tree-based approach		Flow-based approach	
			ILP – OPL	ASP – CMODELS	ILP – OPL	ASP – CMODELS
A	15	58	4.64	1.38	1.92	23.58
		57	-	10.38	1.88	-
B	20	68	10.35	2.30	2.48	226.11
		67	-	18.54	2.37	-
C	25	74	149.94	2.75	3.05	62.70
		73	-	56.24	3.04	-
D	30	76	265.73	2.76	3.76	231.74
		75	-	10.91	4.03	-

Table 2. Experimental results for **RST-DEC** (tree-based approach): program sizes

Problem	# of sinks	k	ILP – OPL		ASP – CMODELS	
			# of variables	# of constraints	# of atoms	# of clauses
A	15	58	11776	72243	36643	110381
B	20	68	15616	115429	41989	126943
C	25	74	19456	168277	46200	140060
D	30	76	23296	230699	50248	152784

Table 3. Experimental results for **RST-DEC** (flow-based approach): program sizes

Problem	# of sinks	k	ILP – OPL		ASP– CMODELS	
			# of variables	# of constraints	# of atoms	# of clauses
A	15	58	8524	10241	109851	362881
B	20	68	11144	13561	139485	463117
C	25	74	13764	16881	168175	560520
D	30	76	16384	20201	196489	656787

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** and the heuristics. With the ILP formulation of **RST-DEC** only (without any heuristics), OPL could not verify in less than 900 sec.s that there is no tree of size $k = 57$ connecting the given sink pins to the source. In the former computation, the program contains 11776 variables and 72243 constraints. With the CP formulation, OPL could not generate a solution for any of the problems in 900 sec.s.; so results for CP are not shown in the tables.

The flow-based ILP formulation significantly improves the computational efficiency, as seen in Tables 1 and 3: for Problem C with $k = 74$, the CPU time decreases to 3.05 sec.s, and the number of constraints to 16881. 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.

3 Haplotype Inference

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. Haplotype inference is the problem of determining the 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 genotypes. For that, researchers have studied a slight modification of the haplotype inference problem, *Haplotype Inference by 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].

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 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 $g[j]$.

Another definition of **HIPP** (and thus **HIPP-DEC**), tuned for ILP, is due to [15]: 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, $g[j] = h_1[j] + h_2[j]$.

3.1 Formulations of HIPP-DEC

Our formulations of HIPP-DEC can be grouped into two depending on how we represent the mappings between genotypes and haplotypes:

Straightforward formulations We introduce auxiliary variables/atoms to represent the mappings between n genotypes and k haplotypes.

Alternative formulations We enforce that each genotype i be explained by haplotypes $2i$ and $2i-1$, and that the number of unique haplotypes is less than or equal to k.

Each group contains formulations of **HIPP-DEC** (and **HIPP**), in each declarative programming paradigm, and according to each group of definitions (due Gusfield, and due Brown & Harrower). In the following we consider the straightforward formulations of problems, in ASP, CP, and ILP, with respect to Gusfield's definitions; these formulations are included in Appendix for convenience. All formulations are available at [9].

3.2 A comparison of the formulations

We compare the formulations of **HIPP-DEC** and **HIPP** both from the point of view of representation, and from the point of view of computational efficiency.

Straightforward formulations relative to Gusfield's definitions In the language of OPL, the given set of genotypes are declared as a matrix g of size $n \times m$, and the set of k haplotypes to be inferred is declared as a matrix h of size $k \times m$. The CP/ILP formulations of **HIPP-DEC** in the language of OPL are shown in Fig.s 1 and 2 in Appendix.

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 $amb(g, j)$. Due to the presence of both classical negation and default negation, these atoms are sufficient to represent the three values of sites. Similarly, we describe the value of the j'th site of a haplotype i by atoms of the form $h(i, j)$. Then the formulations of **HIPP-DEC** in the language of OPL and LPARSE follow from the problem description.

The representations of **HIPP-DEC** in ASP, CP and ILP are declarative, and they are obtained from the problem description systematically in the same way. Therefore, they look similar, but there are some differences due to specifics of the input languages. For instance, in the language of LPARSE, haplotypes and mappings are generated using "choice rules":

```
{h(H,J)} :- haplo(H).
1{s(I,G,H) : haplo(H)}1.
```

Since the language of OPL allows functions, generation of haplotypes and mappings are done in OPL by definitions like

```
dvar int h[1..k,1..m] in 0..1;
dvar int s[1..2][1..n] in 1..k;
```

Although the CP and ILP formulations are both expressed in the language of OPL, there are some differences between them as well. Conditional statements are not allowed in ILP, therefore statements like

```
forall(i in 1..n) forall(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];}
```

in CP can be expressed in ILP as follows:

```
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];
```

Representing **HIPP** *based on the straightforward formulations of* **HIPP-DEC** All three straightforward formalisms have special constructs to solve optimization problems. Let us investigate how these formulations of **HIPP-DEC** discussed 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 $2n \times m$, 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.

From the representation viewpoint, all three paradigms require similar modifications on the existing formulations of **HIPP-DEC** to obtain formulations of **HIPP**.

Adding symmetry breaking constraints and auxiliary atoms/variables Two haplotypes are different if they have a site with different values; otherwise they are identical. To

Table 4. Experimental results for **HIPP-DEC** relative to Gusfield's definition: computation times (CPU sec.s)

Problem	n	m	k	Straightforward formulation			Alternative formulation		
				CP – OPL	ASP – CLASP	ILP – OPL	CP – OPL	ASP– CLASP	ILP – OPL
I0	10	41	16	1.65	3.17	-	0.71	1.01	1.03
			15	-	-	-	20.25	1.01	133.37
I1	10	41	15	0.82	2.63	-	1.25	0.96	1.52
			14	-	-	-	11.35	0.99	1.28
I2	15	48	22	102.29	10.99	-	3.64	5.58	4.05
			21	-	-	-	-	8.74	-
I3	20	54	26	-	608.99	-	58.03	11.07	-
			25	-	-	-	-	109.84	-

improve computational efficiency, we add to the straightforward 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 S1, we can 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. For instance, in the ASP formulation, we add the rules (suggested to us by Martin Gebser)

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

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;
```

Computational efficiency With the formulations discussed above (and presented in Appendix), we solved four problem instances of **HIPP**, randomly chosen amongst the ones tested in [15], using CLASP 1.0.4[7] and ILOG OPL 5.5 with CP-OPTIMIZER 1.1, on a machine with Intel Centrino 1.8GHz CPU and 1 GB of RAM running on Windows XP. For each problem instance, we present results for two instances of **HIPP-DEC**: one with the optimal value of k, and the other with 1 less than the optimal. The optimal value for k was computed by binary search as in [16].

Table 4 compares the computation time (CPU time in sec.s), and Table 5 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). Two sorts of formulations are considered in Table 4: a straightforward representation of Conditions C1–C3 and the alternative

Table 5. Experimental results **HIPP-DEC** relative to Gusfield's definition (straightforward formulation): program sizes

Problem	n	m	k	CP – OPL		ASP – CLASP		ILP – OPL	
				# of var.s	# of constraints	# of var.s	# of constraints	# of var.s	# of rules
I0	10	41	16	4416	1333	88138	87796	-	-
I1	10	41	15	4465	1369	70457	70135	-	-
I2	15	48	22	7551	2299	302462	301770	-	-
I3	20	54	26	11194	3469	612646	611564	-	-

Table 6. Experimental results **HIPP-DEC** relative to Gusfield's definition (alternative formulation): program sizes

Problem	n	m	k	CP – OPL		ASP – CLASP		ILP – OPL	
				# of var.s	# of constraints	# of var.s	# of rules	# of var.s	# of constraints
I0	10	41	16	17351	19695	49071	25432	4110	17641
I1	10	41	15	17387	19695	49089	25432	4110	17641
I2	15	48	22	45017	49175	129734	66452	7410	45571
I3	20	54	26	89387	95605	259899	132322	11360	90201

formulation mentioned above. In Table 4, a dash - indicates that the problem could not be solved in 900 sec.s. For instance, consider Problem I0. A set of 16 haplotypes that explain the given genotypes is computed in 1.65 sec.s using OPL with the CP formulation (straightforward formulation); with the same formulation and the solver, it can not be verified in 900 sec.s that there is no set of 15 haplotypes that explain the given genotypes. In the computation of the former instance, the theory contains 4416 variables and 1333 constraints.[8] The ILP formulations of problems have larger program sizes, mainly due to the larger size of matrices s (since indices cannot be decision variables as in CP); none of the problems can be compiled into a theory in 900 sec.s.

With the alternative definition of **HIPP-DEC** due to [15], with the CP and ASP formulations, the computation time decreases (at least by a factor of 2). For instance, it can be verified that I1 does not have a solution with $k = 14$, in 11.35 (resp. 0.99) sec.s with the alternative CP (resp. ASP) formulation of **HIPP-DEC**; neither OPL nor CLASP can verify this in 900 sec.s with the straightforward formulations. With the alternative ILP formulation, OPL can solve most of the problems (5 out of 8 in Table 4), even though it does not terminate for any of them with the straightforward ILP formulation [9].

With the straightforward **HIPP** formulations, no solution can be found to any problem in 900 sec.s, whereas solutions to some problems can be computed in 900 sec.s with the alternative formulations (Table10 in Appendix).

Adding to the straightforward formulations of **HIPP-DEC**, symmetry breaking constraints S1 does not decrease the timings. On the other hand, adding S2 to the formulations of **HIPP** decreases the computation time for problems smaller than I0–I3: while a problem with 8 genotypes and 9 sites 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, for the same problem, the computation time reduces from 557.42 sec.s to 0.68 sec.s with the ASP formulation.

[8] As Pierre Flener suggested to us, transposing the matrix h in the CP formulation, Fig. 1, generally improves the computational efficiency due to faster propagation: with such a modification the computation of a solution for I0 (resp. I1, I2) takes 0.99 (resp. 0.90, 3.22) sec.s.

4 Conclusion

We studied two challenging problems, wire routing and haplotype inference, in the context of three declarative programming paradigms, ASP, CP, ILP, usually used by different research communities to solve such hard problems. Due to the declarative nature of the formalisms, and the systematic representation of the problems in each formalism, we observed some refreshing similarities: the representations are tolerant to some elaborations (e.g., adding restrictions on lengths of wires in **RST-DEC**, symmetry breaking in **HIPP-DEC**) in the sense of John McCarthy [4].

We observed some differences in the representations due to specifics of the input languages of the solvers. For instance, in the 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. 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 summation, and in ASP one can use a cardinality expression.

Some differences are due to the expressivity of the formalisms. In the ASP representations, one can include recursive definitions, like the definition of reachability in the formulation of **RST-DEC**; in the CP/ILP formulations, we have to enumerate all possibilities (cf. formulations of Steiner tree problem in [14]), or introduce too many auxiliary variables. We have observed that, due to the representation of reachability in ASP, straightforward ASP formulation of **RST-DEC** is more tolerant to some elaborations (e.g., computing a rectilinear tree). Another observation: To express the uniqueness of haplotypes generated so far, in CP/ILP, $2k^2$ auxiliary variables (i.e., `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 the CP/ILP formulations, genotypes are described by $3nm$ atoms. On the other hand, being able to represent functions and matrices in CP/ILP helps us formulate some problems more concisely, as in the formalization of **HIPP-DEC** relative to [15].

In all these three paradigms, adding domain specific heuristics (e.g., in **RST-DEC**, forcing each path connecting the source to a sink to be covered by two circles of a given radius, one around the source and the other around the sink) generally improves computational efficiency; and adding domain independent constraints (e.g., symmetry breaking in **HIPP-DEC**) does not always improve the computational efficiency.

In this study, our goal was not to compare our approaches to wire routing and haplotype inference, with the existing approaches, but to compare ASP, CP, ILP on these two problems. [13] discusses how the tree-based ASP approach to wire routing compares with the existing approaches; they show that variations of wire routing problems can be represented in an elaboration tolerant way, and computational efficiency can be improved with heuristics and when the ASP approach is used within an intelligent algorithm like "hierarchical routing". [16] discusses how various haplotype inference problems, including real problems, can be solved using the alternative formulation of **HIPP-DEC** (with Gusfield's definitions) in ASP; they show that the ASP approach solves the most number of problems compared to the existing HIPP systems based on SAT/ILP/PBO methods.

Acknowledgments We thank Martin Gebser and Benjamin Kaufmann for their help with the use of CLASP and useful suggestions about some ASP formulations, Pierre Flener for his useful suggestions about some CP formulations, and Michael Heusch and Antoine Gargani for their help with the use of OPL. We also thank anonymous reviewers for their constructive comments on an earlier version of the paper.

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 J. on Computing **16**(4) (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. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
7. Hoffman, K.L., Padberg, M.: Encyclopedia of Operations Research and Management Science. Kluwer, Massachusetts (2001)
8. Rossi, F., van Beek, P., Walsh, T.: Handbook of Constraint Programming. Elsevier (2006)
9. ASP-CP-ILP. `http://people.sabanciuniv.edu/esraerdem/asp-cp-ilp.html` (2008)
10. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proc. of the Fifth 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 TCBB **3** (2006) 348–359
16. Erdem, E., Türe, F. In: Proc. of AAAI. (2008) 436-441

Appendix

Straightforward formulations of **HIPP-DEC** with respect to Gusfield's definitions, in CP, ILP and ASP, are shown in Fig.s 1–3. Other formulations are available at [9].

Results of our experiments with the formulations of **HIPP-DEC** and **HIPP**, relative to Brown & Harrower's definitions, are presented in Tables 7–10.

Table 7. Experimental results for **HIPP-DEC** relative to Brown & Harrower's definition: computation times (CPU sec.s)

Problem	n	m	k	Straightforward formulation			Alternative formulation		
				CP – OPL	ASP – CLASP	ILP – OPL	CP – OPL	ASP– CLASP	ILP – OPL
I0	10	41	16	2.16	40.58	-	0.52	4.12	1.02
			15	-	-	-	19.3	4.2	24.29
I1	10	41	15	3.24	27.97	-	1.18	4.2	0.59
			14	-	-	-	10.79	4.2	1.3
I2	15	48	22	93.92	-	-	3.67	23.22	3.34
			21	-	-	-	-	23.03	-
I3	20	54	26	-	-	-	57.89	81.7	-
			25	-	-	-	-	86.11	-

Table 8. Experimental results **HIPP-DEC** relative to Brown & Harrower's definition (straightforward formulation): program sizes

Problem	n	m	k	CP – OPL		ASP – CLASP		ILP – OPL	
				# of var.s	# of constraints	# of var.s	# of rules	# of var.s	# of constraints
I0	10	41	16	3136	411	279230	278786	108176	2570
I1	10	41	15	3095	411	249272	249275	108176	2570
I2	15	48	22	5406	721	910958	909957	-	-
I3	20	54	26	7924	1081	1910134	1909390	-	-

```
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** (wrt Gusfield's definitions) in the language of OPL.

Table 9. Experimental results **HIPP-DEC** relative to Brown & Harrower's definition (alternative formulation): program sizes

Problem	n	m	k	CP – OPL		ASP – CLASP		ILP – OPL	
				# of var.s	# of constraints	# of var.s	# of rules	# of var.s	# of constraints
I0	10	41	16	16839	17235	48405	24612	1240	17231
I1	10	41	15	16839	17235	48405	24612	1240	17231
I2	15	48	22	44159	44855	128585	65012	2370	44851
I3	20	54	26	88079	89125	258165	130162	3800	89121

Table 10. Experimental results for **HIPP** (alternative formulation): computation times

Problem	n	m	Gusfield's formulation			Brown & Harrower's formulation		
			CP – OPL	ASP – CLASP	ILP – OPL	CP – OPL	ASP – CLASP	ILP – OPL
I0	10	41	124.92	2.01	3.06	121.5	22.04	2.55
I1	10	41	140.55	1.77	2.04	131.4	22.67	1.29
I2	15	48	-	13.25	-	-	117.82	-
I3	20	54	-	-	-	-	-	-

```
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** (wrt Gusfield's definitions) in the language of OPL.

```
geno(1..n). % n genotypes
site(1..m). % m sites
haplo(1..k).% k haplotypes
index(1..2).
#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** (wrt Gusfield's definitions) in the language of LPARSE.