More on Wire Routing with ASP

Deborah East and **Mirosław Truszczyński**
Department of Computer Science
University of Kentucky
Lexington KY 40506-0046, USA
email: deast|mirek@cs.uky.edu

Abstract

Wire routing is an important component in the design of very large scale integrated circuits (VLSI). In a recent work Erdem et al. argued that routing problems can be solved by general-purpose answer-set programming solvers. They proposed to view routing as a planning problem in which multiple robots have to determine their paths between matching pins. As in other planning problems, the representation refers to time moments (or a counter of the number of steps taken). We propose a different approach which eliminates the need to represent time. In adition, we develop techniques to limit the search space by eliminating from consideration grid points that are far from the terminal points. We have experimented with our approach using both `smodels` and our program `dcs`. Both programs ran faster on representations proposed here than on the original ones.

Introduction

The design of very large scale integrated (VLSI) circuits requires assistance of computers. One area of VLSI design is the physical layout of wires connecting sets (usualy pairs) of terminal points. This problem is referred to as *routing*. Because of the increasing size of circuits, research in computer aided design (CAD) for VLSI is an active area of research. In a recent paper, Erdem et al. (Erdem, Lifschitz, & Wong 2000) observed that routing can be viewed as a planning problem. Given n pairs of terminal points to be connected, they introduce n robots, one for each pair. The task for the robots is to plan paths between the corresponding pairs of terminals so that these paths do not intersect. Being a planning problem, routing can be solved by answer-set programming (Niemelä 1999; Lifschitz 1999a; 1999b; Marek & Truszczyński 1999). Erdem et al. model the path planning problem using an action language `ccalc` (McCain & Turner 1997). They propose to solve it by first compiling the resulting action theory into propositional logic and, then, by applying a satisfiability checker (they use `relsat` (Bayardo & Schrag 1997) in their work). Their aproach, while conceptually very elegant, is not yet practical. Only rather small routing problems can be solved by methods developed in (Erdem, Lifschitz, & Wong 2000).

In this paper we further study the main observation made by Erdem et al. that routing problems can be solved by answer-set programming. We propose a different way to

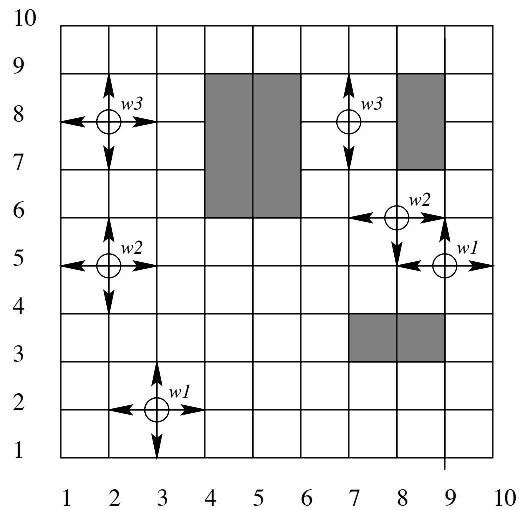

Figure 1: The shaded areas are component regions. The terminal points are indicated by circles. Arrows show which adjacent points can be included in the path.

model the problem. Planning perspective on routing leads to explicit representation of time in the corresponding theories. In our approach, it is not necessary to represent time. Instead, we exploit the structure of the grid to define constraints on paths connecting the corresponding terminal points and to specify how points are included in paths. In particular, each terminal point must belong to exactly one path and exactly one of its adjacent points must also be in the same path. Further, for each internal point on a path exactly two of its adjacent points must also be in the same path.

We do not use counting (time) to move along the path. However, it is advantageous to use distance to limit the search space. We eliminate points for a particular route based on their distance from the terminal points (Niemela 2000) . If the set of paths connecting the terminal points cannot be found, these constraints can be relaxed until a solution is found, or until the entire space has been inluded in the search. This method can also be used with the approach described by Erdem et al. (Erdem, Lifschitz, & Wong 2000).

We have experimented with different representations of the routing problem using both `smodels` (Niemela 1998) and `dcs` (East & Truszczyński 2000). Both programs preformed better on representations proposed in this paper. In particular, we were able to solve problems on grids of size 15×15, while the original approach (Erdem, Lifschitz, & Wong 2000) worked for grids of smaller sizes of 10×10 or so. Despite the progress, scaling to problems of sizes of interest to industry remains a major challenge for answer-set programing approaches. At the very least, however, routing problems emerge as important benchmarks that can be used in development and testing of answer-set programming implementations.

Wire Routing Basics

Physical layout is the last stage in the design of VLSI circuits (Kahng & Robins 1995). This stage has two steps. First, the components are placed on a chip. Second, wires connecting pairs of terminal points are routed so that they do not overlap with each other and with regions occupied by other components. The examples given in this paper and used for experiments and are simplifications of actual wire routing problems. Despite simplifications, in our work we will address some of the key issues arising in routing such as placing multiple wires and restricting individual path lengths. Other issues such as skews (one wire having a much longer path than the others) and delays at terminals will be addressed in future work. Figure 1 shows a simplified chip on a 10×10 grid. The shaded areas are component regions. The terminal points are indicated by circles. The simplified model of wire routing we address in this paper has the following requirements:

1. Wires must not intersect
2. Wires cannot touch components
3. Each wire has a pair of points on the chip which must be connected

Routing Programs

In this section we will present `dcs` and `smodels` programs for routing problems specified by the three requirements given above. `dcs` (East & Truszczyński 2000) combines Horn clauses with constraints and `smodels` (Niemela & Simons 1996) is an implementation of stable logic programming. Both are answer-set programming (ASP) systems.

`dcs`

Figure 2 defines the wire routing problem in the language of `dcs`. We will not give a precise description of the syntax of `dcs`. We will only describe the intuitive meaning of rules comprising the program. The predicates `wire`, `pt`, `block`, `ll`, `upr` and `terminal` are data predicates. They are used to specify the input. The program predicates `inp` and `path` are used in constraints and Horn rules. The intuitive meaning of literal `inp(I,J,W)` is that grid point `(I,J)` belongs to wire `W`. These literals must satisfy rules (1) - (5) of the program. In particular, we have that:

```
idbpred
    inp(pt,pt,wire).
    path(pt,pt,wire).

idbvar
    pt I,J,L,M,P,Q.
    wire  W.

idbrules
1   Select(0,1,W) inp(I,J,W).
2   Select(0,3,L,M;abs(I-L)+abs(J-M)==1)
      inp(L,M,W),inp(I,J,W).
3   Forall(terminal(I,J,W))
      Select(1,1,L,M;abs(I-L)+abs(J-M)==1)
      inp(L,M,W).
4   Forall(block(I,J)) NOT inp(I,J,W).
5   Forall(terminal(I,J,W)) inp(I,J,W).
6   NOT inp(I+1,J,W),inp(I,J+1,W), inp(I,J,W),
      inp(I+1,J+1,W).
7   Forall(terminal(I,J,W))
      Horn inp(I,J,W) -> path(I,J,W).
8   Horn inp(I+1,J,W),inp(I-1,J,W), inp(I,J,W) ->
      path(I,J,W).
9   Horn inp(I,J+1,W),inp(I,J-1,W), inp(I,J,W) ->
      path(I,J,W).
10  Horn inp(I-1,J,W),inp(I,J-1,W), inp(I,J,W) ->
      path(I,J,W).
11  Horn inp(I-1,J,W),inp(I,J+1,W), inp(I,J,W) ->
      path(I,J,W).
12  Horn inp(I+1,J,W),inp(I,J-1,W), inp(I,J,W) ->
      path(I,J,W).
13  Horn inp(I+1,J,W),inp(I,J+1,W), inp(I,J,W) ->
      path(I,J,W).
14  inp(I,J,W) <--> path(I,J,W).
```

Figure 2: A predicate `dcs` program for wire routing.

```
    Forall(ll(I,J,W);P<I-t) NOT inp(P,Q,W).
    Forall(ll(I,J,W);Q<J-t) NOT inp(P,Q,W).
    Forall(upr(I,J,W);P>I+t) NOT inp(P,Q,W).
    Forall(upr(I,J,W);Q>J+t) NOT inp(P,Q,W).
```

Figure 3: Constraints limiting the search space for each wire.

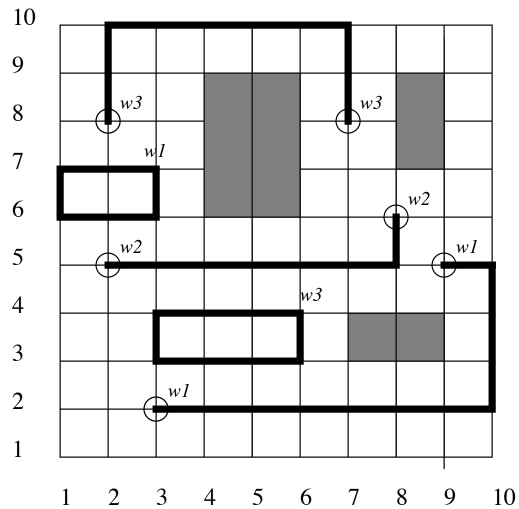

Figure 4: Cycles which are independent to the path can result if Horn clauses are not used in the `dcs` wire routing program.

Rule 1 prevents more than one wire from passing through a point

Rule 2 prevents more than two adjacent points of any point in a path from being included

Rule 3 requires exactly one adjacent point for each terminal point to be included in the path

Rule 4 prohibits all wires from entering a component region

Rule 5 requires that each terminal point is in some path.

Rule 6 prohibits a one block cycle.

In other words, literals `inp(I,J,W)` (if they satisfy these constraints) define collections of paths between terminal points and, possibly, some additional cycles (see Fig. 4). We want to eliminate these cycles as they are not important for solving the routing problem. To this end we use the predicate `path`. The intuitive meaning of literal `path(I,J,W)` is that grid point `(I,J)` belongs to wire `W` *and* it can be reached from a terminal point of `W`. To enforce this intuition we use rules (7) - (13). Finally, we are interested in such initial choice of literals `inp(I,J,W)` for which the literals `inp(I,J,W)` and `path(I,J,P)` coincide (that is, in such a choice in which there are no redundant cycles). Rule (14) enforces this constraint.

The `dcs` program in Figure 2 does not constrain the size of the search space. By including in it the additional rules in Figure 3 we can reduce the search space for each wire to a rectangle determined by its terminal points and the constant `t`. When $t = 0$ the rectangle is the area defined by the lowest row and column values and the highest row and column values in the set of points for a wire. As t increases the size of the rectangle increases in each direction. The predicates `ll` (lower left) and `upr` (upper right) are included in the data file (although they could be compute from the terminal points) and are only used for these constraints.

```
:-2{ path(I,J,W) : wire(W) } , pt(I;J).

1{path(M,N,W):pt(M):pt(N):(abs(I-M)+abs(J-N))=1}1
   :- endpoint(I,J,W),wire(W),pt(I;J).

2{path(M,N,W):pt(M):pt(N):(abs(I-M)+abs(J-N))=1}2
   :- path(I,J,W), not endpoint(I,J,W), wire(W),
   pt(I;J).

:- path(I,J,W), block(I,J), pt(I;J),wire(W).


endpoint(I,J,W) :- terminal(I,J,W).
path(I,J,W) :- terminal(I,J,W).

:- path(I,J,W), path(I+1,J,W),path(I,J+1,W),
   path(I+1,J+1,W), pt(I;J),wire(W).
```

Figure 5: An smodels program for wire routing.

```
:- path(K,L,W), ll(I,J,W), pt(K;L), K < I-t.
:- path(K,L,W), ll(I,J,W), pt(K;L), L < J-t.
:- path(K,L,W), upr(I,J,W), pt(K;L),K > I+t.
:- path(K,L,W), upr(I,J,W), pt(K;L),L > J+t.
```

Figure 6: `smodels` constraints for limiting the search space for wire routing.

smodels

The `smodels` program in Fig. 5 for wire routing is based on similar ideas as the `dcs` program we discussed before. The constraints used to reduce the search space are shown in Fig. 6.

Figure 7 illustrates an example where it is necessary to relax the boundary constraints. In both the `smodels` wire program and the `dcs` wire program, t is a constant that is given a value on the command line of the grounder. If a solution is not found with $t = 0$ (the rectangle formed by terminal points of each wire) then the constraint can be relaxed by increasing the value assigned to t. The layout in Fig. 7 requires $t = 2$.

Experimentation

For experiments, we have implemented a utility program to generate problems which specify chips with components and pairs of wire terminals. The percentage of the area of the chip to be covered with components, and the number of wires and the size of the chip can be specified by the user. The terminal points for the wires are chosen randomly under the following requirements:

1. The terminal points are distinct.
2. No terminal point is in a component area.
3. The pair of terminal points for a wire are at least $\frac{1}{2}$ the size of the chip apart.

We used this utility to generate several test cases (discarding those that could be easily determined as unsolvable and those which required very long paths).

Our experimental results, although still limited, show several interesting aspects of the wire routing problem. First,

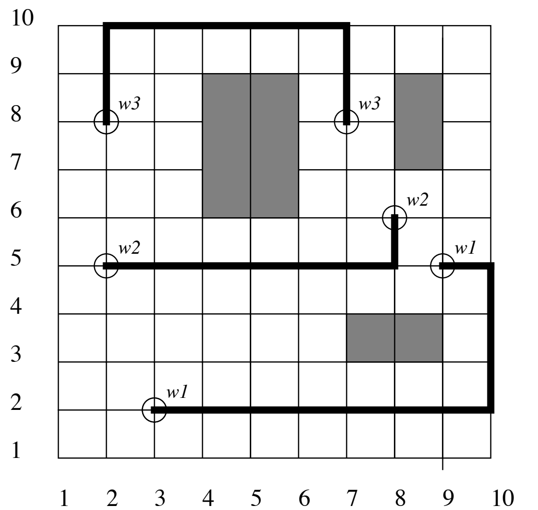

Figure 7: Possible solution for wire route.

Modeling with count 10×10, 4 wires, $\approx 10\%$ blocked			
instance	min length	dcs	smodels
10.1	$f = 10$	25.98	0.30
10.2	$f = 10$	45.28	0.55
10.3	$f = 10$	11.41	0.43
10.4	$f = 10$	2096.28	0.50
10.5	$f = 10$	15.95	0.35
10.6	$f = 10$	20.92	0.24
10.7	$f = 10$	21.20	0.26
10.8	$f = 12$	1961.93	0.26
10.9	$f = 10$	12.33	0.49
10.10	$f = 10$	17.49	0.52

Table 1: f is the minimum path length where a solution was found.

Modeling with count 15×15, 4 wires, $\approx 15\%$ blocked		
instance	min length	smodels
15.1	$f = 11$	0.43
15.2	$f = 15$	2.24
15.3	$f = 15$	1.43
15.4	$f = 13$	0.82
15.5	$f = 13$	1.10
15.6	$f = 15$	30.03
15.7	$f = 15$	1.15
15.8	$f = 15$	5.20
15.9	$f = 15$	2.60
15.10	$f = 12$	0.83

Table 2: f is the minimum path length where a solution was found. Executions were not performed with `dcs` for counting with 15×15 grids.

Modeling w/o count 10×10, 4 wires, $\approx 10\%$ blocked			
instances	relaxation	dcs	smodels
10.1	$t = 1$	0.25	0.06
10.2	$t = 0$	0.04	0.07
10.3	$t = 0$	0.05	0.06
10.4	$t = 0$	0.06	0.09
10.5	$t = 1$	2.05	0.08
10.6	$t = 1$	0.07	0.07
10.7	$t = 1$	2.79	0.08
10.8	$t = 1$	0.47	0.11
10.9	$t = 2$	3102.23	0.20
10.10	$t = 0$	0.06	0.07

Table 3: t is the minimum relaxation where a solution was found.

the problem is a source of good benchmarks for testing ASP implementations and even relatively small instances (on grids of the size 15×15) may be quite difficult. Moreover, input parameters specifying area covered by components and the number of terminal pairs allow the user to control to some degree the difficulty of the problem. We are studying ways to generate instances randomly and expect to locate the phase-transition area where random problems rapidly change from solvable to unsolvable. This work is in progress.

We ran problems we generated both on `dcs` and `smodels`. We also ran them using two different representations: one that required time (or counting the number of steps) and another which did not require counting. The problems we used were difficult for `dcs`, which performed significantly worse than `smodels`. In particular, we observed a big variability in the performance of `dcs` much bigger than in that of `smodels`. We attribute this behavior to limited lookahead used by `dcs` (full lookahead is used by `smodels`). However, using full lookahead in `dcs`, while possible in theory, is at present time not practical — performance deteriorates even further. This implies that additional work on

Modeling w/o counting 15×15, 4 wires, $\approx 15\%$ blocked			
instance	relaxation	dcs	smodels
15.1	$t = 2$	***	1.34
15.2	$t = 0$	0.62	0.19
15.3	$t = 0$	3.50	0.26
15.4	$t = 2$	***	0.47
15.5	$t = 0$	0.85	0.21
15.6	$t = 0$	15.90	0.24
15.7	$t = 0$	612.90	0.18
15.8	$t = 0$	3.03	0.24
15.9	$t = 0$	0.61	0.21
15.10	$t = 0$	0.67	0.23

Table 4: t is the minimum relaxation where a solution was found. *** stopped after one hour.

improved implementation of lookahead in `dcs` is necessary. The variability in the running time found for `smodels` is also larger than one might want (the ratio of the largest execution time to the smallest one in the case of grids of size 15×15 is over 60 for programs involving counting, and 7 for the other representation). This demonstrates the difficulty of routing problems.

For all instances executed by both `smodels` and `dcs` the approach that did not rely on counting was more efficient in time and space. Comparison of execution times is given in Tables 1,2,3,4. The times are given for solving the instances with the indicated constraints. These constraints limit the search space. Without these constraints, times would be much larger. The values for t and f in Tables 1,2,3,4 are those for the tightest constraints where a solution exists. The programs not using counting are more efficient in terms of size of their groundings (see Table 5). That seems to be a key factor in better performance when processing them (as opposed to those that involve counting). `smodels` performs very well on problems we generated. It can solve instances substantially larger than those discussed in (Erdem, Lifschitz, & Wong 2000). However, the comparison is not quite fair as restrictions on the allowed area for the wires were not used in (Erdem, Lifschitz, & Wong 2000). This study is still to be done.

Neither approach guarantees that a solution with minimum total path length is found. The counting approach ensures, though, that the solution found minimizes the length of the longest path. This is not the case for the approach that does not use counting (see Fig. 8).

We have shown here that our approach for encoding wire routing is a viable alternative to the planning approach. Theories produced in this manner are smaller than those using counting for the encoding. The smaller size of the theories is vital as we scale up the size of the problems.

In conclusion, wire routing is a source of good benchmarks for ASP implementations. Wire routing problems discussed in this paper show that a specific representation used in modeling a problem (counting or no counting, in our case) significantly affects the performance of the solver. Thus, the

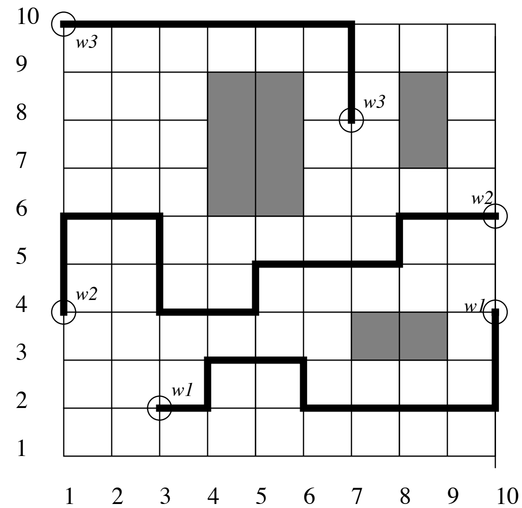

Figure 8: Paths do not find shortest routes.

`smodels`				
Size	w/o cnt		cnt	
	Atoms	Rules	Atoms	Rules
10×10	1385	≈ 2884	5389	≈ 12695
15×15	3050	≈ 6629	16254	≈ 36540
`dcs`				
10×10	800	≈ 3148	9000	≈ 166057
15×15	1800	≈ 8602	29100	≈ 762721

Table 5: Differences in size of theories for both counting and non-counting approach. The constraints are $f = 10$ for 10×10, $f = 15$ for 15×15, $t = 0$.

issue of programming methodology for ASP systems is a very important one. Finally, wire routing allows us to experiment with domain knowledge and its effect on the performance. Adding restrictions on the area where wires can run is an example of domain-specific knowledge that improves the performance greatly.

References

Bayardo, R., and Schrag, R. 1997. Using csp look-back techniques to solve real-world sat instances. In *Proceedings of IJCAI*, 203–208.

East, D., and Truszczyński, M. 2000. DATALOG with Constraints. In *Proccedings of the Seventeenth National Conference on Artificial Intelligence(AAAI-2000)*.

Erdem, E.; Lifschitz, V.; and Wong, M. 2000. Wire routing and satisfiability planning. In *Proceedings CL-2000*.

Kahng, A., and Robins, G. 1995. *On optimal interconnections for VLSI*. Boston MA: Kluwer Academic Publishers.

Lifschitz, V. 1999a. Action languages, answer sets, and planning. In Apt, K.; Marek, W.; Truszczyński, M.; and Warren, D., eds., *The Logic Programming Paradigm: a 25-Year Perspective*. Springer Verlag. 357–373.

Lifschitz, V. 1999b. Answer set planning. In *Proceedings of the 1999 International Conference on Logic Programming*, 23–37. MIT Press.

Marek, V., and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In Apt, K.; Marek, W.; Truszczyński, M.; and Warren, D., eds., *The Logic Programming Paradigm: a 25-Year Perspective*. Springer-Verlag. 375–398.

McCain, N., and Turner, H. 1997. Causal theories of action and change. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*. MIT Press.

Niemela, I., and Simons, P. 1996. Efficient implementation of the well-founded and stable model semantics. In *Proceedings of JICSLP-96*. MIT Press.

Niemela, I. 1998. Logic programs with stable model semantics as a constraint programming paradigm. In *Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning*, 72–79.

Niemelä, I. 1999. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence* 25(3-4):241–273.

Niemela, I. 2000. Personal communication.