Comparing ASP and CP on Four Grid Puzzles

Mehmet Çelik, Halit Erdoğan, Fırat Tahaoğlu, Tansel Uras, and Esra Erdem

Faculty of Engineering and Natural Sciences
Sabanci University, Istanbul / Turkey

Abstract

We study two declarative programming languages namely Answer Set Programming (ASP) and Constraint Programming (CP) on four grid puzzles: Akari, Kakuro, Nurikabe, and Heyawake. We represent these problems in both formalisms in a systematic way and compute their solutions using ASP system Clasp and CP system Comet. We compare the ASP approach with the CP approach both from the point of view of knowledge representation and from the point of view of computational time and memory.

1 Introduction

Grid Puzzles have been studied in various areas of AI, and these studies have led to interesting insights and useful approaches for solving them as well as some real-life problems. In this paper, we study the representation of four grid puzzles Akari, Kakuro, Nurikabe, and Heyawake in Answer Set Programming (ASP) and in Constraint Programming (CP). The objective is to compare these declarative programming paradigms both from the point of view of representation and from the point of view of computational efficiency on these puzzles. We prepared a website[1] where you can find an extended version of the paper, full ASP and CP encodings, CP encodings for other solvers, puzzle instances, comparison of ASP solvers (with other grounders), ongoing work and further results that do not fit into this paper.

Comparing ASP and CP from the point of view of representation 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 formalism in a similar way, and try to understand how easy/hard it is to express these concepts in these declarative paradigms.

Comparing ASP and CP from the point of view of computational efficiency We compute the solutions to the grid puzzles using the ASP system Clasp[2] and the CP system Comet[3]. We compare these computations in terms of CPU time and program size.

[1] `http://krr.sabanciuniv.edu/projects/gridpuzzle/`
[2] `http://www.cs.uni-potsdam.de/clasp/`
[3] `http://www.comet-online.org/`

Proceedings of the 16th International RCRA workshop (RCRA 2009):
Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion
Reggio Emilia, Italy, 12 December 2009

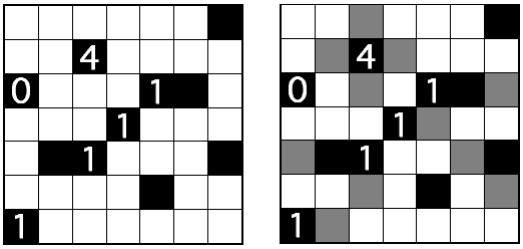

Figure 1: A sample Akari puzzle and its solution. The black cells with numbers are called "hints". Gray cells are the light bulbs.

In the literature, there are several attempts to solve the puzzles that we are interested in as well as the empirical works to compare several declarative programming paradigms. For instance, [1] studies the representations of some grid puzzles in ASP including Heyawake and Nurikabe. In [2] the authors compare ASP, CP and ILP on Wire Routing problems and Haplotype Inference problems. [10, 11] presents CP formulations of some grid puzzles. In addition, in [7], the authors study the comparison of ASP and CP on some puzzles. Similarly, [4] presents some experimental results for comparing Constraint Logic Programming and Answer Set Programming on some NP-complete problems.

In the rest of the paper, we show the representations of each puzzle in the language of Lparse (grounder for the ASP solver Clasp) and in the language of Comet, show the results of our experiments with Clasp and Comet using these formulations and point out the weaknesses and strengths of each approaches.

2 Akari

Akari, also known as Light-Up, uses a rectangular grid of black cells and white cells. Hints (integers) are placed in black cells. Figure 1 is an example of an initial Akari board and its solution. The player solves puzzles via placing light bulbs in the white boxes according to following rules:

A1 Light bulbs are permitted to be placed at any white square. A hint (numbered black square) indicates how many light bulbs are next to it, vertically or horizontally.

A2 Each light bulb illuminates from itself to a black square or the outer frame in its row and column, and every white square must be illuminated.

A3 No light bulbs should illuminate each other.

Given an Akari problem, deciding whether a solution exists is NP-complete [9].

2.1 An ASP Formulation of Akari

Input: The initial board is described with two types of atoms:

- `hint(Row,Column,Value)` is used to specify hints. `Row,Column` specifies the location of the hint[4], `Value` specifies the value (constraint) of the hint. For example, in Fig. 1, there is a hint at the bottom left corner with the value 1 and represented by `hint(1,1,1)`.
- `black(Row,Column)` specifies the black cells (including the hints). All non-black cells are considered to be white. For example, there is a black cell in Figure 1 at the top right corner and represented by `black(1,7)`.

Output: The output is described by atoms of the form `bulb(Row,Column)`, that means the cell (`Row,Column`) has a bulb in it.

The constraints are forced as follows in ASP:

A1 Light bulbs are permitted to be placed at any white square. A hint (numbered black square) indicates how many light bulbs are next to it, vertically and horizontally.

```
C{bulb(V1,H1) : adjacent(V,H,V1,H1) : white(V1,H1)}C :- hint(V,H,C).
```

A2-A3 Each light bulb illuminates from itself to a black square or the outer frame in its row and column, and every white square must be illuminated.

```
1{bulb(V,H) : reachable(V,H,V1,H1) : verticalIndex(V) :
horizontalIndex(H)}2

:- white(V1,H1),verticalIndex(V1),horizontalIndex(H1).
```

A cell is vertically (horizontally) reachable from another, if it is vertically (horizontally) adjacent to it, or reachable from any of its vertical (horizontal) adjacent cells.

[4]In all of the puzzle grids in this paper, the cell on the bottom left corner has the location $(1,1)$, and the cell on the top right corner has the location (m,n), where m is the number of columns and n is the number of rows.

2.2 A CP Formulation of Akari

Input: The input is given as a series of tuples for black cells, hints and vertical and horizontal blocks. The hint inputs are tuples of 3: row, column, and the value, while the black cells are identified by their rows and columns. The vertical (or horizontal) blocks are identified by tuples of 3, the vertical (or horizontal) index of the block and the horizontal (or vertical) beginning and end indices of the block. So, all vertical and horizontal blocks are given in input, according to these a neighborhood set is constructed for each cell, which contains reachable cells from that cell.

Output: The output is an integer matrix `board`. For every (i, j), $board(i, j)$ is mapped to either 0 or 1, where $board(i, j) = 1$ indicates that the cell (i, j) has a bulb.

Rules: The rules are enforced via constraints as follows:

A1-I Light bulbs are permitted to be placed at any white square.

```
forall(i in 1..nbB){
  //read row and column index from input
  m.post(board[r,c] == 0);}
```

A1-II A hint (numbered black square) indicates how many light bulbs are next to it, vertically and horizontally.

```
forall(i in 1..nbH){
  //read row, column and constraint from input
  m.post(board[r,c] == 0);
  m.post((board[r,c-1] + board[r,c+1] +
  board[r-1,c] + board[r+1,c]) == v);
}
```

A2-A3 Each light bulb illuminates from itself to a black square or the outer frame in its row and column, and every white square must be illuminated.

```
for all block x{
  //that block should have at most 1 bulb
  m.post(sum(i in x) board[i.r, i.c] <= 1);
   forall(row in RowRange)
    forall(column in ColumnRange)
     if(the cell is white){
      //every white cell should be lighted
      m.post(sum(j in neighborhood[row,column])
                 board[j.r, j.c] >= 1);}}
```

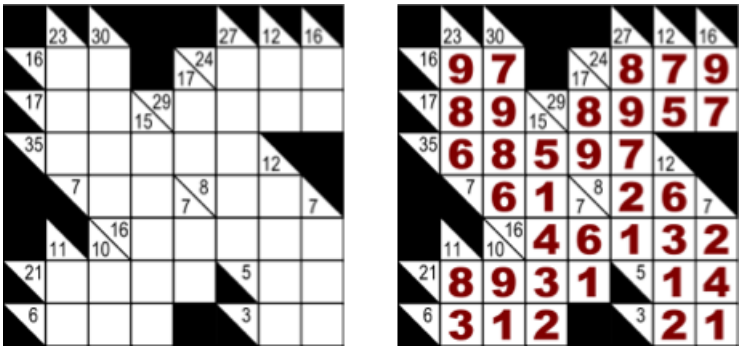

Figure 2: A sample Kakuro puzzle and its solution.

3 Kakuro

Kakuro uses a rectangular grid of black and white cells. Black cells may contain hints (integers) as in Akari. The number below the diagonal divider is the hint for cells below, the number above the diagonal divider is the hint for cells to the right. Figure 2 shows an example and solution for a Kakuro problem. The goal is to fill the white cells while satisfying the following rules:

K1 Each white cell should contain a single value between 1-9.

K2 All numbers in a continuous block of white cells must be pairwise different.

K3 The sum of a continuous block of white cells in horizontal (or vertical) direction must be equal to the hint given in the black cell to the left (above).

Deciding whether there is a solution for a Kakuro instance is NP-complete [6]

3.1 An ASP Formulation of Kakuro

Input: The initial board is described with three types of atoms:

- `hHint`(`Row`, `Column`, `Sum`, `Extent`) is used to specify horizontal hints. `Row`, `Column` specifies the location of the hint, `Sum` specifies the sum constraint of the hint and `Extent` specifies how many cells the hint covers. For example, in Figure 2, there is a horizontal hint which requires 3 cells and with the constraint value 6 on the bottom left corner of the grid and represented by `hHint(1,1,6,3)`.

- `vHint`(`Row`, `Column`, `Sum`, `Extent`) is used to specify vertical hints, similar to the horizontal one.

- `black(Row, Column)` specifies the black cells (including the hints). All non-black cells are considered to be white. For example, the cell $(1, 5)$ is black and represented by `black(2,4)`.

Output: The output is described by atoms of the form `cellVal(Row, Column, Value)`, meaning the cell `Row, Column` is assigned `Value` that ranges between 1 and 9.

The rules are enforced via constraints and are as follows:

K1 For each white cell, there must be exactly one `cellVal` atom

```
1{cellVal(R,C,Val) : val(Val)}1 :- not black(R,C), row(R),
column(C).
```

K2 For any hint block and any two different cells in it, the value of the cells cannot be the same

```
:-  vHint(R, C, Sum, E), cellVal(R1, C, Val),
    cellVal(R2,C,Val), row(R1; R2),
    R < R1, R1 < R2, R2 <= R + E, val(Val).
% Same rule applies for the horizontal hint (hHint)
```

K3 The enforcement of the sum constraint is a bit harder. Basically, for each hint (horizontal or vertical) we sum the values of the cells starting from the farthest cell in the hint block all the way to the first cell of the hint. Then the total sum is forced to be equal to the value of the hint. Step by step:

1. The end points for the hints are determined and the sums up to these points are set to 0.

```
horSum(R,C+E,0) :- hHint(R,C,Sum,E).
verSum(R+E,C,0) :- vHint(R,C,Sum,E).
```

2. The horizontal and vertical sums are computed all the way to the hints

```
verSum(R-1,C,Sum+Val)
    :- not black(R,C), verSum(R,C,Sum),
    cellVal(R,C,Val), sum(Sum), val(Val).
% Same rule applies for the horizontal sum (horSum)
```

3. The total sum cannot be different than the value of the hint

```
:- vHint(R,C,Sum,E), verSum(R,C,Sum2),
   sum(Sum2), Sum != Sum2.
% Same  rule  applies for the horizontal hint (hHint)
```

3.2 A CP Formulation of Kakuro

Input: The input is given as a series of tuples for vertical hints, horizontal hints and black cells. The hint inputs are tuples of 4: row, column, sum and extent of the hint, while the black cells are identified by their rows and columns. The input for vertical and horizontal hints are processed in the program to form an array called `hints` where a `hint` consists of a set of cells associated with it and its sum constraint.

Output: The output is an integer matrix called `board`, representing the puzzle. The values in each cell range between 0-9 where 0 means the cell is black and 1-9 is the fill value of a white cell.

For each hint the following function is called with the solver, the hint itself and the board:

```
function void postHint(Solver<CP> m, block bl, var<CP>{int}[,]
board)
```

This function contains the following parts:

K1 All white cells are non-black (their domains are 1-9)

```
forall (a in bl.cells)
  m.post(board[a.r,a.c]>=1);
```

K2 All cells in the hint's set should contain different values

```
m.post(alldifferent(all(a in bl.cells) board[a.r,a.c]));
```

K3 The sum of the values of the cells in the hint's set should satisfy the sum constraint of the hint

```
m.post(sum(a in bl.cells) board[a.r,a.c] == bl.sumC);
```

Also, all of the black cells are forced to be 0 in the input reading phase.

4 Nurikabe

Nurikabe is played on a rectangular grid of cells where some cells are numbered. The challenge is to paint some of the cells black while obeying the following set of constraints and definitions.

N1 The "walls" are made of connected adjacent white cells in the grid of cells.

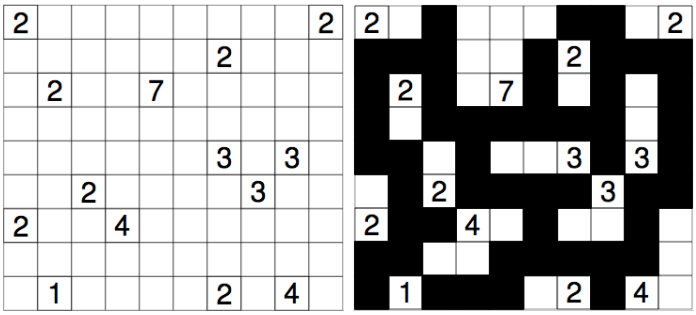

Figure 3: A sample Nurikabe puzzle and its solution.

N2 At the start of the puzzle, each numbered cell defines (and is one block in) a wall, and the number indicates how many white cells the wall must contain. The solver is not allowed to add any further walls beyond these.

N3 Walls shall not connect to each other.

N4 Any cell which is not a block in a wall is part of "the maze" – meaning that is a black cell.

N5 The maze must be a single orthogonally (vertically or horizontally) contiguous whole.

N6 The maze is not allowed to have any "rooms" – meaning that the maze may not contain any 2x2 squares of black cells.

Deciding whether there is a solution to Nurikabe is NP-complete [8].

4.1 An ASP Formulation of Nurikabe

Problem representation in ASP is nearly identical with the representation in [1]. We try some modifications in the representation but they did not affect the search of the ASP solver, and produce negligible changes in the computational time and efficiency.

4.2 A CP Formulation of Nurikabe

Input and Output: The input is represented as two list of values. Tuples for the rows, columns to express the position of the numbered cells and the values of the numbered cells in the grid. The output is a binary matrix board where $[i, j]$ is 1 if the cell is painted white and 0 otherwise.

In the CP formulation, all constraints related to neighborhood relations are represented as follows:

N2 At the start of the puzzle, each numbered cell defines a wall, and the number indicates how many white cells the wall must contain.

```
forall(k in 1..NumberedCellsRange)
  m.post((sum(i in 1..r, j in 1..c) board[i,j]==k))
  == Number[k]);
```

N3 Walls shall not connect to each other.

```
forall(i in 2..r-1) //without matrix borders
  forall(j in 2..c-1)
    m.post(board[i,j] != 0 =>(
     (board[i+1,j] == 0 || board[i+1,j] == board[i,j])
     && (board[i,j+1] == 0 || board[i,j+1] == board[i,j])
     && (board[i,j-1] == 0 || board[i,j-1] == board[i,j])
     && (board[i-1,j] == 0 || board[i-1,j] == board[i,j])));
```

N6 The maze is not allowed to have any "rooms" (meaning that the maze may not contain any 2x2 squares of black cells).

```
forall(i in 1..r-1)
  forall(j in 1..c-1)
    m.post(board[i,j] != 0 || board[i+1,j] != 0
    || board[i,j+1] != 0 || board[i+1,j+1] != 0);
```

$N1$ and $N4$ are definitions needed for the implementations of the other constraints. The most challenging part of the CP formulation of this puzzle is to represent $N5$. This constraint forces all the black cells in the solution to be reachable to each other. The CP representation of "reachability" is not as easy as in ASP, since COMET does not allow transitive closure. Therefore, we have not represented this constraint explicitly in CP, but we do change the search mechanism: we ensure that COMET goes over each computed solution and checks whether the solution satisfies the reachability constraint by performing a depth first search from a black cell to other black cells. If we encounter all the black cells in the search than we say it satisfies $N5$.

5 Heyawake

Heyawake is played on a rectangular grid whose cells are white or numbered, like in Nurikabe. Furthermore, different from Nurikabe, the grid is divided into rooms, as shown in Figure 4. The goal is, like in Nurikabe, to paint some cells black Figure 4, but obeying a different set of rules:

H1 Painted cells may never be orthogonally connected.

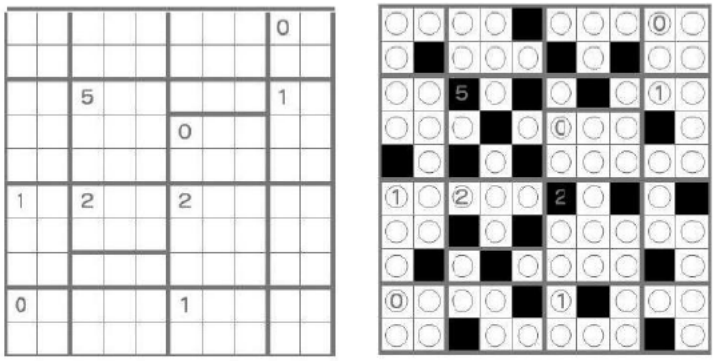

Figure 4: A sample Heyawake puzzle and its solution.

H2 All white cells must be interconnected.

H3 A number (also known as room black cell size) indicates exactly how many painted cells there must be in that particular room.

H4 A room which has no number may contain any number of painted cells (including the possibility of zero cells).

H5 Where a straight (orthogonal) line of connected white cells is formed, it must not contain cells from more than two rooms. In other words, any such line of white cells which connects three or more rooms is forbidden.

Deciding whether there is a solution to Heyawake is NP-complete [5].

5.1 An ASP Formulation of Heyawake

We modify the ASP representation of Heyawake in [1] for a better computational efficiency.

Input: The initial board is described with two types of atoms:

- `room(N,R1,C1,R2,C2)` indicates the boundaries of the room N as the rows between $R1 - R2$ and the columns between $C1 - C2$. For example, in Figure 4, the first room in the top left corner of the grid is represented as `room(1,1,9,2,10)`.
- `has(N,V)` indicates that the room N must contain V black cells. For example, the room in the bottom left corner of the grid is the third room and has a black cell size of 0 and represented by `has(3,0)`.

Output: Atoms of the form $black(R, C)$ which indicates that the cell (R, C) is black.

We have observed that, in the ASP representation of Heyawake in [1] the most time consuming part of the representation is the formulation of the constraint $H5$: The atoms that represent length of the vertical (or horizontal) path between two cells, take six arguments, and thus require a lot of time and space consumption since the grounding results in huge number of rules. Therefore, we have reformulated the constraint $H5$. In this representation: *nroom(X,Y)* describes that the rooms X and Y are neighbor rooms and *inroom(C,R,T)* represents that the cell (C, R) is in room T. In addition, the atoms of the form $vconnected(C, R, C1, R1)$ (resp. $hconnected(C, R, C1, R1)$) represent a vertical (resp. horizontal) straight path between (C, R) and $(C1, R1)$. Using these atoms, we reformulate the constraint $H5$ as below. *It is not allowed that two white cells are connected by a orthogonal path and included in rooms that are not neighbors.* The representations of other constraints are the same as in [1].

```
% H5: It is not allowed that two white cells are straight connected
% and included in non neighbor rooms.

:- vconnected(C,R,C1,R1), inroom(C,R,T), inroom(C1,R1,T1), not
nroom(T,T1), num(T;T1).

:- hconnected(C,R,C1,R1), inroom(C,R,T), inroom(C1,R1,T1), not
nroom(T,T1), num(T;T1).
```

5.2 A CP Formulation of Heyawake

Input: The input is very similar to the input of ASP formulation except that, instead of atoms, we obtain tuples of integers for the boundaries and black cell sizes of the rooms. We preprocess the input and have a matrix `NeighborRooms` where `NeighborRooms[R1,R2]` determines whether the rooms $R1$ and $R2$ are neighbor or not.

Output: The output is a binary matrix that represents the solution of the puzzle, where the 1s denote the black cells and 0s denote the white cells.

In the CP representation, we have a binary decision variable `Grid[i,j]` which is 1 if the cell (i, j) in the solution is black; it is 0 if the cell is white. Using this decision variable we represent the constraints as follows:

H1 Painted cells can never be orthogonally connected.

```
forall(i in 1..noofRows)
  forall(j in 1..noofColumns){
    if((i-1) > 0)
      m.post(Board[i,j] + Board[i-1,j] < 2);
    if((i+1) <= noofRows)
      m.post(Board[i,j] + Board[i+1,j] < 2);
    if((j-1) > 0)
      m.post(Board[i,j] + Board[i,j-1] < 2);
    if((j+1) <= noofColumns)
      m.post(Board[i,j] + Board[i,j+1] < 2);
    }
```

H3 & H4 A number (also known as room black cell size) indicates exactly how many painted cells there must be in that particular room. A room which has no number may contain any number of painted cells

```
forall(room in 0..noofRooms)
  if(RoomBlackCellSize[room] > -1)
    m.post((sum(cell in RoomContains[room])
      (Board[cell.r, cell.c]==1))==RoomBlackCellSize[room]);
```

H5 A straight (orthogonal) line of connected white cells is formed, it must not contain cells from two different rooms.

```
forall(i in 1..noofRows)
  forall(j1 in 1..noofColumns)
    forall(j2 in j1..noofColumns)
      if(AdjRooms[CellInRoom[i,j1],CellInRoom[i,j2]] != 1)
        m.post( (Board[i,j1] + Board[i,j2] == 0) =>
        ((sum(k in j1..j2) Board[i,k]) > 0)  );
```

As in Nurikabe, we have not represented the constraint $H1$ since it requires reachability. Instead, COMET goes over the computed solutions (with a depth first search fashion) and determine whether there is a solution which satisfies reachability.

6 Experimental Results and Discussion

In this section, we present the experimental results for the formulations described in the previous sections. The ASP representations are tested on the ASP solver CLASP (with default settings) version 1.2.1 (LPARSE as grounder) and the CP representations are tested on the CP solver COMET (with default settings) version 2. Table 1 shows the results for each puzzle instance. For each puzzle instance we show the computational time that is consumed by each solver[5]. We compare the program sizes, in terms of number of

[5] All CPU times are in seconds, for a workstation with a 2.1GHz Intel Core2 Duo T6570 processor and 2,96GB RAM, running MS Windows XP 32 bit.

atoms and number of rules for CLASP and number of variables and number of constraints for COMET. In addition, we compare the search strategies of the solvers by means of "number of choices" and "number of conflicts" for CLASP, and number of choices and number of fails for COMET. The number of choices, conflicts, and fails will give us insights about how much/less the solvers search while finding the solution.

When we look at the results for the Akari instances, we can see that both solvers do not have much difficulty solving these puzzles. For the small instances, CLASP performs better; whereas, for the large instances COMET performs better. One can see that number of choices made both by COMET and CLASP are outnumbered by the choices made in other puzzles. In addition, few number of encountered conflicts and fails indicate that both solvers solve this problem with few number of search steps.

For the Kakuro instances, COMET finds solutions by doing less search, spending less computational time in comparison with CLASP. For these instances, we can say that COMET outperforms CLASP. The strength of COMET for these instances can be explained by the usage of the `alldifferent` propagator which is known to be very efficient constraint. As regards the ASP representation, we have recursive definitions for the summations which results in huge program after grounding. The summations are handled more easily in CP with an iterative approach.

On the other hand, for the Nurikabe and Heyawake instances, CLASP performs much better than COMET. There are instances for which CLASP finds solutions in a couple of seconds where COMET cannot find a solution in 10 minutes. Most reasonable explanation for that is the unnatural handling of reachability in CP. In ASP, we represent reachability easily since it allows recursive transitive closure definitions. On the other hand, we could not find an effective reachability representation in CP. Not posting reachability as a constraint but going over the found solutions consumes a lot of time.

As interesting future work, we would like to represent reachability in constraint programming via enumerating all possible paths with all possible lengths between two cells. For that, we keep decision variables for each path between each two cells with each length. As a result, we will force that there exists a path of any length between two cells; by doing that we force reachability of two cells. However, our initial test results using GECODE[6], does not seem to be impressing from the point of view of computational efficiency. According to the results of our discussions with Helmut Simonis, we believe that implementing propagators for reachability can be another way for representing reachability in CP, and this will be more effective to solve Heyawake and Nurikabe. Furthermore, we can use the CP solvers that includes built-in propagators for reachability, like ECLIPSe[7]. We also try

[6] `http://www.gecode.org`
[7] `http://87.230.22.228/`

to use the GECODE library for graph concepts, cp(graph) [3], which allows representation of reachability in GECODE; but, we could not manage to use it properly.

We have also made a comparison between two ASP solvers, CLASP and CMODELS, using LPARSE as the grounder. The results can be found in Table 2.

References

[1] M. Cayli, A. G. Karatop, E. Kavlak, H. Kaynar, F. Ture, and E. Erdem. Solving challenging grid puzzles with answer set programming. In *Proc. of ASP*, pages 175–190, 2007.

[2] E. Coban, E. Erdem, and F. Ture. Comparing ASP, CP, ILP on two challenging applications: wire routing and haplotype inference. In *Proc. of LaSh*, 2008.

[3] G. Dooms, Y. Deville, and P. Dupont. Cp(graph): Introducing a graph computation domain in constraint programming. In *Proc. of CP*, pages 211–225. Springer-Verlag, 2005.

[4] A. Dovier, A. Formisano, and E. Pontelli. An experimental comparison of constraint logic programming and answer set programming. In *AAAI*, pages 1622–1625. AAAI Press, 2007.

[5] M. Holzer and O. Ruepp. *Fun with Algorithms*, chapter the troubles of interior design: a complexity analysis of the game heyawake, pages 198–212. Springer Berlin / Heidelberg, 2007.

[6] G. Kendall, A. J. Parkes, and K. Spoerer. A Survey of NP-Complete Puzzles. *International Computer Games Association Journal*, 31(1):13–34, 2008.

[7] T. Mancini, D. Micaletto, F. Patrizi, and M. Cadoli. Evaluating ASP and Commercial Solvers on the CSPLib. *Constraints*, 13(4):407–436, 2008.

[8] B. McPhail. Nurikabe is NP-Complete. NW Conference of the CCSC, Poster Session, October 2004. Student poster.

[9] B. McPhail. Light Up is NP-Complete. Unpublished manuscript, February 2005. Available from `www.cs.umass.edu/~mcphailb/papers/#lightup`.

[10] H. Simonis. Sudoku as a constraint problem. In *Proc. of fourth Int. Works. on Modelling and Reformulating Constraint Satisfaction Problems*, 2005.

Table 1: Experimental results for the puzzle instances for Akari (a1...a10), Kakuro (k1...k10), Nurikabe (n1...n10), and Heyawake (h1...h10). No result showed if the solver cannot find a solution in 10 minutes.

		Clasp					Comet				
Puzzle Instance	Grid size	time (sec.)	# of atoms	# of rules	# of choices	# of conflicts	time (sec.)	# of vars.	# of cons.	# of choices	# of fail
a1	7x7	0.206	1337	2778	2	1	0.541	81	868	3	1
a2		0.223	1353	2834	0	0	0.431	81	866	1	0
a3		0.208	1348	2804	0	0	0.441	81	1138	4	0
a4		0.195	1255	2524	6	4	0.451	81	1307	14	1
a5		0.184	1135	2198	0	0	0.419	81	887	1	0
a6	10x10	1.146	3214	8596	0	0	0.705	144	4012	7	0
a7		0.993	2929	7430	0	0	0.629	144	3316	1	0
a8		0.749	2440	5627	7	4	0.63	144	4397	10	2
a9	14x14	4.996	5079	14497	8	4	1.29	256	13846	9	4
a10		4.922	5016	14256	11	7	1.33	256	13841	13	6
k1	8x8	0.686	8541	55200	830	383	0.461	49	101	5	4
k2	12x10	1.309	15600	103707	1844	831	0.418	99	214	19	13
k3	10x14	1.971	18093	120454	6001	2673	0.429	117	221	31	38
k4	12x12	1.801	18588	124093	3794	1685	0.433	121	248	24	22
k5	14x12	2.425	21593	145902	9846	5118	0.453	143	280	316	557
k6	12x16	2.756	24596	166487	5061	2007	0.458	165	352	43	42
k7	20x12	4.717	30583	207548	6874	2587	0.455	209	427	35	19
k8	16x16	5.865	32520	221332	6566	2778	0.455	225	461	68	79
k9	20x14	6.298	35528	242768	12380	5088	0.535	247	521	270	368
k10	32x22	32.365	87705	612210	38499	10798	2.492	651	1327	6135	8729
n1	5x5	0.156	1505	5060	59	20	0.29	25	70	79	84
n2		0.171	1509	5100	32	11	0.305	25	71	45	45
n3		0.156	1510	5064	176	57	0.317	25	70	322	402
n4		0.171	1514	5103	49	22	0.298	25	71	31	27
n5	7x7	0.89	5298	20010	213	60	0.318	49	143	118	152
n6		1.203	5286	19809	1112	582	7.022	49	140	159349	224779
n7		0.828	5294	19941	285	70	0.342	49	142	757	1069
n8		0.875	5294	19942	554	124	0.548	49	142	4769	7159
n9	10x10	7.64	21991	85376	1168	304	-	-	-	-	-
n10	12x12	23.406	42899	179413	1152	218	-	-	-	-	-
h1	6x6	0.239	3085	7705	22	15	0.386	36	160	296	396
h2		0.26	3220	7843	18	5	0.455	36	162	204	235
h3		0.234	3150	7796	25	13	0.575	36	171	647	776
h4		0.239	3064	7696	75	40	2.555	36	164	6200	6837
h5	7x7	0.355	6522	15333	12	5	0.616	49	298	342	440
h6		0.375	5792	14503	55	25	2.468	49	251	2273	2668
h7	10x10	1.271	26923	64353	36	19	4.491	100	731	3443	4392
h8		1.514	24820	62240	186	53	56537	100	725	59972	87180
h9		1.94	26014	63546	447	164	-	-	-	-	-
h10	15x15	10.268	134472	328372	1090	139	-	-	-	-	-

Table 2: A comparison of two ASP solvers, CLASP and CMODELS on puzzle instances for Akari (a1...a10), Kakuro (k1...k10), Nurikabe (n1...n10), and Heyawake (h1...h10). No result showed if the solver did not find a solution in 15 minutes.

	CLASP					CMODELS			
Instance ID	#of atoms	#of rules	#of variables	#of constraints	time	#of atoms	#of clauses	#of rules	time
h1	26923	64353	53607	155512	1.161	50279	169937	64424	2.197
h2	24643	61976	55527	159822	1.154	48211	165528	62061	2.505
h3	26497	63994	59606	172914	1.490	50844	177441	64076	2.293
h4	26235	64236	57914	167676	0.948	50844	177754	64076	2.334
h5	24820	62240	57984	168849	1.366	50811	175246	62321	2.081
h6	26011	63549	59593	172495	0.933	51810	180110	61321	2.581
h7	24737	62061	57113	166917	1.247	51038	175531	62140	2.871
h8	134471	328373	297077	864230	6.076	250461	878613	328567	14.039
h9	136928	331015	292316	850539	7.281	251624	879489	331198	22.436
h10	135282	329307	300901	876217	10.161	252744	889134	329502	11.443
a1	1072	2006	217	180	0.07	1915	3789	2237	0.08
a2	976	1716	199	125	0.06	1579	2961	1897	0.07
a3	887	1458	181	113	0.05	1232	2123	1597	0.05
a4	990	1840	209	140	0.06	1531	2837	2022	0.07
a5	1369	2871	307	84	0.10	3120	6662	3233	0.12
a6	1361	2848	282	0	0.10	3112	6655	3211	0.12
a7	2652	6446	546	189	0.62	5859	12416	7128	0.66
a8	2446	5624	520	94	0.58	4977	10254	6198	0.61
a9	5795	17701	1281	47	4.42	13607	29370	19296	4.36
a10	4505	11734	942	417	2.75	8554	17914	12720	2.87
k1	8409	55068	8324	16133	0.51	4300	19804	55356	0.63
k2	15413	103520	15004	27998	1.11	7451	34869	104088	1.26
k3	17893	120254	18270	36048	1.73	9615	43975	120862	1.77
k4	18383	123888	17608	32328	1.51	8580	40496	124552	1.65
k5	21383	145692	32196	71992	2.26	19452	84346	146572	2.56
k6	24351	166242	29655	62110	2.44	16748	74440	167210	2.63
k7	30297	207172	33698	64643	4.23	17204	79636	208356	3.93
k8	32219	221031	35805	69127	4.5	18398	84933	222287	4.62
k9	35209	242449	43524	89528	5.39	24086	107923	243881	5.09
k10	87175	611580	131458	283847	27	76508	336340	615452	20.29
n1	20968	84620	85653	15540	2.28	81098	313366	84719	304.72
n2	20976	84880	83709	196230	6.79	79157	282991	84979	274.81
n3	20976	84883	83357	162430	1.28	74880	257094	84982	26.86
n4	20976	84881	83624	193032	1.71	76914	269182	84980	106.55
n5	20984	85144	81309	159233	2.18	73101	257120	85243	33.02
n6	20972	84750	84810	198051	1.56	88934	328047	84849	232.56
n7	20980	85016	82285	165014	2.17	74058	253353	85115	17.85
n8	103396	437439	456628	1145286	100.66	na	na	na	na
n9	103400	437716	453016	1086001	877.44	na	na	na	na
n10	103408	438278	446638	1018565	519.3	na	na	na	na

[11] H. Simonis. Kakuro as a constraint problem. In *Proc. seventh Int. Works. on Constraint Modelling and Reformulation*, 2008.