

Kakuro as a Constraint Problem

Helmut Simonis

CrossCore Optimization
London
UK

Abstract. In this paper we describe models of the logic puzzle Kakuro as a constraint problem with finite domain variables. We show a basic model expressing the constraints of the problem and present various improvements to enhance the constraint propagation, and compare alternatives using MILP and SAT solvers. Results for different puzzle collections are given. We also propose a grading scheme predicting the difficulty of a puzzle for a human and show how problems can be tightened by removing hints.

1 Introduction

Kakuro is one the many logical puzzles popularized by the Japanese company Nikoli [27]. Table 1 shows a small scale example taken from [26] and its solution.

	23	30		24	27	12	16
16				17			
17			29				
			15				
35						12	
	7			8			7
				7			
	11	10					
21					5		
6					3		
	23	30		24	27	12	16
16	9	7		17	8	7	9
17	8	9	29	15	8	9	5
			15				
35	6	8	5	9	7	12	
	7			8			7
				7	6	1	2
				16			
11	10		4	6	1	3	2
21	8	9	3	1	5	1	4
6	3	1	2		3	2	1

Table 1. Example Problem and Solution

The puzzle is described by the following rules:

1. The puzzle uses a rectilinear grid of black and white cells. Black cells may contain hints (integer numbers). 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.

2. The task is to enter numbers from 1 to 9 into the white cells satisfying the following constraints.
3. The sum of a continuous block of white cell in horizontal (or vertical) direction must be equal to the hint given in the black cell to the left (above).
4. All numbers in a continuous block of white cells must be pairwise different.

The grid size of the puzzle can vary, the largest instance in our evaluation has 124 rows and 90 columns. A *valid* Kakuro puzzle has solutions, a *well posed* problem admits a single solution. An interesting feature of logical puzzles is that the solution should be deduced with a finite set of deduction rules, without using search. A characterisation of the difficulty of a puzzle instance is the rule set required to solve it.

We show in this paper that all instances from commercial sources used in the tests can be solved by constraint propagation with a generalized arc consistent (GAC) version of the *alldifferent-sum* constraint, which combines rules 3 and 4 of the problem description for a set of variables and a simple redundant constraint modelling the interaction of row and column constraint pairs. We see that a naive model using *alldifferent* and *sum* constraints is not sufficient, and see how the full constraint propagation can be achieved without writing a specific new global constraint. The constraint programming approach closely resembles the way most humans solve the puzzle by hand.

We also present (naive) models using MILP (mixed integer linear programming) and a PseudoBoolean model mapped to a SAT solver, which show that even for relatively small problem sizes the problem is not trivial.

2 Related Work

Kakuro has become widely popular in the wake of the Sudoku craze that swept country after country in recent years[26]. While the problems are related, their models show significant differences. While Sudoku is mainly concerned with the propagation of the *alldifferent* constraint and the interaction of multiple such constraints [18, 9, 13, 7, 1], we find that the challenge for Kakuro is the interaction of an *alldifferent* constraint with the *sum* constraint over the same variables. The *alldifferent-sum* constraint is not found in the global constraint catalog [2], but a more general constraint *weighted-partial-alldiff* could be used. The PhD thesis of S. Thiel [20] describes GAC propagation for part of this constraint. Implementing this global constraint in ECLiPSe for the sole purpose of this analysis seemed excessive.

On the other hand, the chosen method in this paper is re-using ideas first applied to crossword puzzles [21, 12]. We use the *propia* library of ECLiPSe [28] to obtain a generalized arc consistent *alldifferent-sum* constraint by simple program annotation. We show that for the problem considered this is competitive with state of the art constraint libraries written in C++.

3 Initial Model

The initial model for solving the Kakuro puzzle with finite domain variables is fairly straightforward. For each white cell, we introduce a finite domain variable which ranges over values from 1 to 9. For each hint over a continuous block of variables we introduce an *alldifferent* and a *sum* constraint. Slightly more formally, a Kakuro puzzle is defined by a tuple $\langle G, H \rangle$ with G a set of cell locations, and H a set of hints given as tuples $\langle v, I \rangle$ with a positive integer v and a set $I \subset G$. The model then consists of variables x_i

$$\forall_{i \in G} : x_i \in [1, 9]$$

and two types of constraints. The *alldifferent* constraint states that a set of variables must be pairwise different

$$\forall_{\langle I, v \rangle \in H} : \text{alldifferent}(\{x_i | i \in I\})$$

The *sum* constraint states that the sum of the variables in a hint must be equal to a given integer value.

$$\forall_{\langle I, v \rangle \in H} : \sum_{i \in I} x_i = v$$

Together with a search routine, this defines a model of the problem. We use the built-in *search* procedure of the *ic* library of ECLiPSe 5.10 with default values

```
search(L,0,input_order,indomain,complete,[])
```

and impose a timeout of 300 seconds. Note that we don't use any clever method for variable or value selection. As we want to solve the puzzle without search at all, the search routine only serves as a backstop.

A generic technique which helps the reasoning for many puzzles is *shaving*[10], already used for Sudoku in [18]. Before starting search it tests for each variable and each value in its domain if the assignment would lead to a failure. If that is the case, then the value can be safely removed, strengthening the constraint propagation. This is applied recursively until no more domain reduction can be achieved. For logical puzzles the time required for shaving usually is well spent, as it removes possible dead-ends and helps with propagation and variable selection.

Most human puzzle solvers frown upon the use of shaving as a deduction method, since it tests out values and therefore "performs search". On the other hand, it is a very effective technique which only uses polynomial time.

4 Evaluation

We test our model on a selection of puzzles from different sources:

big A single very large puzzle instance from Nikoli [22].

mix Some puzzles from the Penpa Mix puzzle collections of Nikoli [24].

giants Another puzzle collection from Nikoli, containing large scale examples [25].

kakuro A special issue from Nikoli on Kakuro [23].

jnp A collection of number puzzles [6] which contains some Kakuro instances.

suzuki Another collection of different puzzle types [19] containing a number of Kakuro puzzles. Note that some puzzle instances have multiple solutions, they are not used in our evaluation.

The problem sizes range from very small (9x9 grid with 36 variables) to quite large (124 rows, 90 columns and 8339 variables). The experiments for all systems were run on a laptop with 2GHz Pentium M processor and 1Gb of memory under Linux.

Table 2 shows the result for the basic model with a recursive shaving routine, which iterates shaving until saturation is reached (i.e. no more elements can be removed). The search is limited by a timeout of 300 seconds. *K* indicates the number of instances in the set, *Setup* the percentage of problems solved at problem setup, *Shave* the percentage of problems solved after shaving and *Total* the percentage of problems solved after search. Average and maximal execution times for the solved instances are given, as well as average and maximal backtrack counts for the solved instances.

Set	X	Y	K	Setup	Shave	Total	Avg Time	Max Time	Avg Back	Max Back
big	124	90	1	0.00	0.00	100.00	258.09	258.09	577710.00	577710
giants	32	22	46	0.00	54.35	89.13	10.01	287.37	26776.51	760019
giants	32	42	6	0.00	33.33	83.33	6.25	15.53	13705.80	36702
giants	32	46	1	0.00	0.00	100.00	68.92	68.92	161256.00	161256
jnp	9	9	8	0.00	100.00	100.00	0.01	0.01	0.00	0
jnp	10	10	8	0.00	87.50	100.00	0.03	0.07	3.88	31
jnp	12	12	8	0.00	50.00	100.00	0.04	0.08	4.75	32
kakuro	10	14	39	0.00	100.00	100.00	0.02	0.04	0.00	0
kakuro	16	16	44	0.00	81.82	100.00	0.13	2.26	150.34	6210
kakuro	32	22	18	0.00	61.11	100.00	1.03	8.82	2413.11	31867
mix	12	12	12	0.00	100.00	100.00	0.01	0.02	0.00	0
mix	16	16	70	0.00	87.14	100.00	0.08	0.58	31.11	1615
mix	32	22	8	0.00	50.00	100.00	0.85	2.14	1019.75	3565
suzuki	20	12	44	0.00	97.73	100.00	0.04	0.12	0.43	19
All			313	0.00	80.51	98.08	2.63	287.37	6403.28	760019

Table 2. Basic Model with Shaving

None of the problems are solved by propagation alone, and even after shaving only about 80% of instances are solved. Note that 6 of the larger instances were not solved within the given time limit. The contribution of shaving is significant. Without it, only 90% of the problems are solved within the time limit (15 unsolved instances).

5 MILP Model

To check if the constraint approach is competitive, we have tried both (naive) MILP and SAT formulations of the problem and tested them with the *eplex* library in ECLiPSe [17] and with Minisat+ [5], a Pseudo-Boolean problem pre-processor for Minisat [4].

The MILP model uses 0/1 integer variables y_{ij} which indicate if cell i takes value j , j ranging from 1 to 9.

$$\forall_{i \in G}, \forall_{j \in [1,9]} : y_{ij} \in \{0, 1\}$$

We then have to restrict the variables for the same cell to state that exactly one of the y_{ij} variables must be one, i.e. one of the values from 1 to 9 must be taken. This is expressed with a set of equations:

$$\forall_{i \in G} : \sum_{j \in [1,9]} y_{ij} = 1$$

The next constraint type states that cells in the same block must contain different values, i.e. that value j can only be taken once for all cells in the index set of a hint.

$$\forall_{\langle I, v \rangle \in H}, \forall_{j \in [1,9]} : \sum_{i \in I} y_{ij} \leq 1 \quad (1)$$

The last constraint type expresses the arithmetic constraints, stating that the sum over all cells in a hint index set must be equal to the hint value v . Since our basic model uses 0/1 integers, we need a rather lengthy linear form of the constraint:

$$\forall_{\langle I, v \rangle \in H} : \sum_{i \in I} \sum_{j \in [1,9]} j * y_{ij} = v$$

The model does not have a real objective function, as we are only looking for a (the unique) feasible solution. A dummy objective function

$$\min \sum_{i \in G} \sum_{j \in [1,9]} y_{ij}$$

is used in our test runs. Alternatives do not seem to have a major impact. Table 3 shows results for the default Coin-OR[8] solver (CLP/CBC) in *eplex*[17] with a timeout of 300 seconds. 76 (mostly large) instances can not be solved within the timeout, only 76% of all instances are solved.

6 Pseudo Boolean Model

We can re-use the MIP model as the basis for a Pseudo-Boolean model, which is expanded into a SAT model. We use the same 0/1 variables, and only have to

Group	X	Y	K	Nr Vars	Solved	Avg Time	Max Time
big	124	90	1	n/a	n/a	n/a	n/a
giants	32	22	46	4465.17	10.87	98.81	241.31
giants	32	42	6	8742.00	0.00	n/a	n/a
giants	32	46	1	9423.00	0.00	n/a	n/a
jnp	9	9	8	366.75	100.00	0.85	4.65
jnp	10	10	8	572.62	100.00	3.41	10.40
jnp	12	12	8	820.12	100.00	9.54	31.18
kakuro	10	14	39	745.38	100.00	1.02	6.90
kakuro	16	16	44	1487.45	95.45	29.81	151.25
kakuro	32	22	18	4351.00	27.78	153.23	301.24
mix	12	12	12	744.00	100.00	1.10	6.25
mix	16	16	70	1494.00	92.86	23.03	248.53
mix	32	22	8	4341.38	12.50	3.94	3.94
suzuki	20	12	44	1357.77	100.00	17.00	287.30
All			313	2122.33	75.71	20.78	301.24

Table 3. MILP Model Overview

transform the less or equal constraint (1) into a greater or equal constraint, as required by the data format for Minisat+.

$$\forall \langle I, v \rangle \in H, \forall j \in [1, 9] : \sum_{i \in I} -y_{ij} \geq -1$$

We rely on the automated translation of the equations and inequality constraints into clausal form in Minisat+, and do not use any of the possible control parameters. As we only require a feasible solution, there is no objective function.

Results for the combination of Minisat+ and Minisat2.0 [4, 5] are shown in table 4, which report the time required to find a first solution. Proving that the solution is unique does not significantly increase execution times.

Even by only using the default settings, the results are very consistent. All problems (up to 695000 clauses) except the largest one (5.2 million clauses) are solved within 300 seconds, with an average solution time of nearly 18 seconds. A solution for the “big” instance is not found in 10 hours.

7 Improving Propagation

It is disappointing that our basic finite domain constraint model was not able to solve all problems, but a simple reflection shows that missing constraint propagation is to blame. Consider a block of five cells with a sum of 15. This is modelled as

```
[X1,X2,X3,X4,X5] :: 1..9,
alldifferent([X1,X2,X3,X4,X5]),
X1+X2+X3+X4+X5 = 15
```

Set	X	Y	K	Solved	Restart	Conflict	Avg Dec	Max Dec	Avg Time	Max Time
big	124	90	1	0.00	n/a	n/a	n/a	n/a	n/a	n/a
giants	32	22	46	100.00	15.13	91793.39	292493.80	817837	51.64	173.06
giants	32	42	6	100.00	17.00	165484.33	695334.00	920827	178.34	254.92
giants	32	46	1	100.00	17.00	163183.00	778007.00	778007	204.01	204.01
jnp	9	9	8	100.00	5.50	1437.75	4282.75	7412	0.43	0.57
jnp	10	10	8	100.00	8.38	5198.00	14062.75	28060	1.34	2.10
jnp	12	12	8	100.00	10.38	11475.12	29130.00	48752	2.75	4.91
kakuro	10	14	39	100.00	7.59	3939.18	11064.38	23566	1.30	2.38
kakuro	16	16	44	100.00	11.43	18389.02	52642.30	135559	5.24	13.06
kakuro	32	22	18	100.00	14.56	61504.61	224880.67	340253	41.65	74.16
mix	12	12	12	100.00	7.50	3714.08	10850.58	21390	1.11	1.80
mix	16	16	70	100.00	10.97	15185.57	46195.66	101888	5.15	11.06
mix	32	22	8	100.00	15.25	79101.00	264553.25	365393	40.65	67.36
suzuki	20	12	44	100.00	9.73	9082.39	30027.75	53245	3.60	7.58
All			313	99.68	11.15	31196.45	103787.51	920827	17.87	254.92

Table 4. SAT Model Overview

There is no propagation from the *alldifferent* constraint, and the bounds calculated for the variables in the equality constraint

$$\sum_{k \in I} x_k = N$$

are

$$\bar{x}_i = N - \sum_{k \in I, k \neq i} x_k$$

$$\underline{x}_i = N - \sum_{k \in I, k \neq i} \bar{x}_k$$

which evaluate to 11 and -21, and therefore do not constrain the variables either.

But as we know that the values must be different, we can compute an upper bound of $5 = 15 - (1 + 2 + 3 + 4)$, i.e. we can remove values 6 to 9 from all domains. In order to achieve this propagation, we can either just pre-compute the domain restrictions as a redundant model, or consider the *alldifferent-sum* constraint as a global, generalized arc consistent constraint. We try the first, simpler approach in the next section, and then consider the *alldifferent-sum* constraint.

7.1 Domain Reduction

How do we know which values we can remove for which constraint? We can precompute this (with a simple FD constraint program) by considering every constraint of a given arity and fixed total sum. In 33 cases we can solve the *alldifferent-sum* constraint by this domain reduction and an GAC *alldifferent* constraint. In a further 31 cases, we can reduce the domains, without solving

the constraint completely. This leaves 55 cases where no reduction is possible. Fortunately, these cases only occur sporadically in the given problem instances. The constraint of arity 9 is a special case. The only possible sum is 45, which is reached by any permutation of nine different values, so again a GAC *alldifferent* constraint is sufficient.

By applying the domain reductions as a first step before setting up any other constraint, we dramatically improve performance, as shown in table 5.

Set	X	Y	K	Setup	Shave	Total	Avg Time	Max Time	Avg Back	Max Back
big	124	90	1	0.00	100.00	100.00	1.76	1.76	0.00	0
giants	32	22	46	4.35	100.00	100.00	0.08	0.18	0.00	0
giants	32	42	6	0.00	100.00	100.00	0.17	0.22	0.00	0
giants	32	46	1	0.00	100.00	100.00	0.18	0.18	0.00	0
jnp	9	9	8	50.00	100.00	100.00	0.01	0.02	0.00	0
jnp	10	10	8	25.00	100.00	100.00	0.01	0.04	0.00	0
jnp	12	12	8	0.00	100.00	100.00	0.01	0.02	0.00	0
kakuro	10	14	39	76.92	100.00	100.00	0.01	0.04	0.00	0
kakuro	16	16	44	27.27	100.00	100.00	0.03	0.05	0.00	0
kakuro	32	22	18	5.56	100.00	100.00	0.07	0.11	0.00	0
mix	12	12	12	91.67	100.00	100.00	0.01	0.02	0.00	0
mix	16	16	70	27.14	100.00	100.00	0.02	0.08	0.00	0
mix	32	22	8	12.50	100.00	100.00	0.07	0.13	0.00	0
suzuki	20	12	44	2.27	100.00	100.00	0.02	0.05	0.00	0
All			313	26.52	100.00	100.00	0.04	1.76	0.00	0

Table 5. Basic Model with Removed Values

Now a quarter of the problems are solved at setup, and all instances are solved after shaving, without calling the search routine.

Note that the same reduction can be applied to the SAT and MILP models. Table 6 shows the result for Minisat+. It now solves all problem instances with an average time of 2 seconds, the maximal time required is 434 seconds for the “big” instance with 2.1 million clauses. The results for MILP are also improved, but the model still only solves 95% of all problem instances within 300 seconds.

7.2 GAC *alldifferent-sum*

Reducing the initial domain of the variables is only a first step in improving the reasoning for the *alldifferent-sum* constraint. Ideally, we want to treat it as a global constraint and enforce generalized arc consistency in its propagation, i.e. for each constraint all unsupported values are removed as soon as possible. But writing a new global constraint for just this purpose seems excessive. Is there another way of achieving generalized arc consistency for our problem?

Using ECLiPSe [28], we can use the *propia* [12] library for generalized propagation. This is based on the observation that the number of feasible solutions

Set	X	Y	K	Solved	Restart	Conflict	Avg Dec	Max Dec	Avg Time	Max Time
big	124	90	1	100.00	16.00	87910.00	1006558.00	1006558	434.58	434.58
giants	32	22	46	100.00	4.65	2022.70	8786.11	50232	1.80	7.61
giants	32	42	6	100.00	7.00	3434.17	25951.83	51882	6.40	11.37
giants	32	46	1	100.00	7.00	2734.00	25118.00	25118	6.04	6.04
jnp	9	9	8	100.00	1.12	57.00	180.25	413	0.06	0.08
jnp	10	10	8	100.00	1.88	205.88	643.12	2908	0.12	0.35
jnp	12	12	8	100.00	2.25	290.25	1068.62	4031	0.18	0.42
kakuro	10	14	39	100.00	1.05	11.74	44.23	526	0.06	0.12
kakuro	16	16	44	100.00	1.82	194.64	745.39	4935	0.22	0.85
kakuro	32	22	18	100.00	3.28	554.50	2941.78	9599	0.86	1.84
mix	12	12	12	100.00	1.00	6.83	35.00	136	0.06	0.08
mix	16	16	70	100.00	1.53	134.59	570.49	6116	0.20	0.80
mix	32	22	8	100.00	3.12	638.25	3146.00	12087	0.94	2.52
suzuki	20	12	44	100.00	2.32	315.55	1091.64	7616	0.23	0.93
All			313	100.00	2.39	818.58	5775.44	1006558	1.99	434.58

Table 6. SAT Model with Removed Values

to each *alldifferent-sum* constraint is limited. For two variables and sum 3 there are only two possible solutions, [1,2] and [2,1]. For nine variable (sum 45), there are $9! = 362880$ possible solutions.

Note that in ECLiPSe it is not necessary to generate the tables up-front. We can use the “*infers*” notation of *propia* shown below to state that we want to use some program as a GAC constraint[15].

```
alldifferent_sum(L,N):-
    sumup(L,Sum),
    (eval(Sum) #= N,
     alldifferent(L),
     labeling(L)) infers ac.
```

For performance reasons it is essential to generate the constraints in the correct sequence, by increasing arity. This reduces the domains of the variables early, so that we don’t have to consider all possible combinations for the large arity constraints.

Alternatively, if we generate the tables for all constraints, we can use a *table* constraint or multiple arc-consistent *element* constraint to achieve the same propagation.

To reduce overall execution time, we perform the initial domain restriction before starting to set-up the constraints. As this removes inconsistent values very rapidly, we reduce the amount of work left for the more complex constraints. To reduce computation time further, we can set up the *alldifferent* constraint and *sum* constraints of the basic model before setting up the GAC version. This again removes some inconsistent values before the more complex reasoning is

started. Table 7 shows the results for our improved ECLiPSe model, combining all techniques discussed above.

Set	X	Y	K	Setup	Shave	Total	Avg Time	Max Time	Avg Back	Max Back
big	124	90	1	100.00	100.00	100.00	2.73	2.73	0.00	0
giants	32	22	46	100.00	100.00	100.00	0.17	1.28	0.00	0
giants	32	42	6	100.00	100.00	100.00	0.27	0.35	0.00	0
giants	32	46	1	100.00	100.00	100.00	0.28	0.28	0.00	0
jnp	9	9	8	100.00	100.00	100.00	0.01	0.01	0.00	0
jnp	10	10	8	100.00	100.00	100.00	0.01	0.03	0.00	0
jnp	12	12	8	87.50	100.00	100.00	0.03	0.06	0.00	0
kakuro	10	14	39	100.00	100.00	100.00	0.02	0.05	0.00	0
kakuro	16	16	44	100.00	100.00	100.00	0.04	0.07	0.00	0
kakuro	32	22	18	94.44	100.00	100.00	0.11	0.22	0.00	0
mix	12	12	12	100.00	100.00	100.00	0.02	0.05	0.00	0
mix	16	16	70	100.00	100.00	100.00	0.04	0.09	0.00	0
mix	32	22	8	100.00	100.00	100.00	0.09	0.12	0.00	0
suzuki	20	12	44	97.73	100.00	100.00	0.04	0.09	0.00	0
All			313	99.04	100.00	100.00	0.07	2.73	0.00	0

Table 7. Combined Model

The interesting result is that all but three of the instances considered are now solved just by initially propagating the constraints, neither shaving nor search is required. We will consider the three remaining problem instances in section 8.

7.3 Alternative Models

To compare our ECLiPSe solution with an efficient finite domain solver in C++, we did run our test cases using the Kakuro program written by C. Schulte and M. Lagerkvist in the Gecode[16] system. After fixing two small problems, we obtained the results shown in table 8.

The Gecode program generates *regular* constraints[11] based on finite automata for the *alldifferent-sum* constraints, which provide GAC propagation. The average solving time for both systems is nearly identical, but the individual solving times vary significantly. The last three columns show the *min*, *average* and *max* ratio of the Gecode time to the combined model in ECLiPSe.

Another model using the *gcc* constraint with cost [14] was suggested by M. Carlsson[3] and tested with Sicstus Prolog. The propagation does not achieve GAC for the *alldifferent-sum* constraint.

8 Redundant Constraints

Only three problems remain unsolved after initial constraint propagation with an GAC *alldifferent-sum* constraint. Table 9 shows the relevant part of an unsolved

Set	X	Y	K	Setup	Total	Time		Ratio Gecode/ECLiPSe		
						Avg	Max	Min	Avg	Max
big	124	90	1	100.00	100.00	0.64	0.64	0.23	0.23	0.23
giants	32	22	46	100.00	100.00	0.19	0.89	0.10	1.18	4.27
giants	32	42	6	100.00	100.00	0.19	0.44	0.24	0.71	1.63
giants	32	46	1	100.00	100.00	0.40	0.40	1.38	1.38	1.38
jnp	9	9	8	100.00	100.00	0.00	0.01	0.50	0.83	1.00
jnp	10	10	8	100.00	100.00	0.03	0.08	0.67	2.36	4.00
jnp	12	12	8	87.50	100.00	0.09	0.48	0.50	4.29	24.00
kakuro	10	14	39	100.00	100.00	0.00	0.03	0.50	0.68	1.50
kakuro	16	16	44	100.00	100.00	0.08	0.47	0.13	1.97	9.33
kakuro	32	22	18	94.44	100.00	0.13	0.57	0.20	1.29	6.33
mix	12	12	12	100.00	100.00	0.00	0.01	0.50	0.83	1.00
mix	16	16	70	100.00	100.00	0.06	0.49	0.14	1.73	12.25
mix	32	22	8	100.00	100.00	0.14	0.36	0.60	1.34	3.60
suzuki	20	12	44	97.73	100.00	0.08	0.43	0.25	2.01	8.67
All			313	99.04	100.00	0.08	0.89	0.10	1.67	24.00

Table 8. Gecode Model Results

puzzle after propagation (instance jnp-142). If a white cell contains a number, that value has been fixed by propagation. Otherwise, the remaining domain is shown. We are interested in the top four unsolved cells on the right hand, coloured in red, which we call $A \in \{8, 9\}$, $B \in \{8, 9\}$, $C \in \{5, 8\}$, $D \in \{6, 9\}$. We can remove value 8 from A. If A is 8, then B must be 9 (alldifferent), and C must be 5 (alldifferent). The assignment of B to 9 removes 9 from the domain of D (alldifferent), leaving 6. But then $C + D = 11$, not 14 as required by the horizontal sum. This causes a failure. Note we can only deduce this through the interaction of two horizontal and two vertical *alldifferent-sum* constraints.

Instead of defining specific pattern to capture such redundant constraint reasoning, we again use generalized propagation as suggested in [15] to capture the interaction of row and column constraints. The program remains quite simple. For every quadruple of unsolved, intersecting pairs of horizontal and vertical constraints we impose the *interact* constraint shown below, which provides a restricted form of path consistency. L is the set of variables occurring in the constraint.

```

interact(L,L1,N1,L2,N2,L3,N3,L4,N4):-
    (alldifferent_sum(L1,N1),
     alldifferent_sum(L2,N2),
     alldifferent_sum(L3,N3),
     alldifferent_sum(L4,N4),
     lableing(L)) infers ac.

```

By adding these redundant constraints to the combined model we solve the remaining three problems at constraint setup, requiring neither shaving nor search.

3	9	2	6	1	45	
1	26	8	9	3	6	42
5	8	9	6	10	3	7
2	3	27	3	7
		11			.89	.89
21	6	5	2	1	4	3
9						
5	7	6	1	14
	12			11	.5	.6
4	7		22	9
		12			7...	.8.
3	2	1	7	2	1	4
9			24			
1	3	2	7	7	2	5
				10		
6	29	5	
			
2	15	4	
			
			.89	78.	789	
		23	...	
			
			
			.89	

Table 9. jnp-142: State after Propagation

9 Grading Puzzles

Most of the puzzle collections that we considered group the problems in multiple grades. This grading is typically based on how difficult the designer finds to solve the puzzle. As this is not only dependent on the techniques used, but also on the particular order in which the methods are applied, this is a highly subjective measure, which often leads to frustration for puzzle users. We propose a more objective measure of puzzle difficulty based on a constraint model which mimics human solving techniques. As a first step, the domain reduction discussed in section 7.1 is used. In addition we use a forward checking version of *alldifferent*, and the usual *sum* constraint. This models “obvious” propagation steps when values are assigned. After this initial propagation, we impose the GAC *alldifferent-sum* constraints by increasing arity. This sequence is also normally used by humans, as it reduces the number of individual cases to be considered. Table 10 shows result of an evaluation with this model. After setting up the initial model, about 26% of all instances are solved, increasing to 99% when setting up all GAC constraints. This grading roughly corresponds to the grade assigned by the designers, shown in column *Grade*, with some notable exceptions. The “medium” mix-12-12 problems for example are actually easier than the “easy” problems of the same size. The results also show that GAC is really required for hard instances, even for the largest arity of the *alldifferent-sum* constraint.

10 Eliminating Hints

For logical puzzles we are always interested if they are given in their minimal form, i.e. if they contain redundant hints. Reducing hints will make the puzzle

Set	Grade	X	Y	K	Setup	P2	P3	P4	P5	P6	P7	P8	P9
big	hard	124	90	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	100.00
giants	hard	32	22	46	4.35	4.35	6.52	43.48	54.35	65.22	82.61	86.96	100.00
giants	hard	32	42	6	0.00	0.00	0.00	0.00	33.33	83.33	83.33	83.33	100.00
giants	hard	32	46	1	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	100.00
jnp	easy	9	9	8	50.00	50.00	62.50	100.00	100.00	100.00	100.00	100.00	100.00
jnp	medium	10	10	8	25.00	25.00	37.50	62.50	75.00	100.00	100.00	100.00	100.00
jnp	medium	12	12	8	0.00	0.00	0.00	25.00	50.00	75.00	87.50	87.50	87.50
kakuro	easy	10	14	24	91.67	91.67	100.00	100.00	100.00	100.00	100.00	100.00	100.00
kakuro	medium	10	14	12	66.67	66.67	91.67	91.67	100.00	100.00	100.00	100.00	100.00
kakuro	hard	10	14	3	0.00	0.00	33.33	100.00	100.00	100.00	100.00	100.00	100.00
kakuro	easy	16	16	5	80.00	80.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
kakuro	medium	16	16	22	27.27	27.27	54.55	72.73	100.00	100.00	100.00	100.00	100.00
kakuro	hard	16	16	17	11.76	11.76	11.76	29.41	58.82	76.47	94.12	94.12	100.00
kakuro	hard	32	22	18	5.56	5.56	11.11	44.44	77.78	88.89	88.89	94.44	94.44
mix	easy	12	12	8	87.50	87.50	100.00	100.00	100.00	100.00	100.00	100.00	100.00
mix	medium	12	12	4	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
mix	medium	16	16	30	33.33	33.33	46.67	70.00	96.67	96.67	100.00	100.00	100.00
mix	hard	16	16	40	22.50	22.50	32.50	77.50	85.00	92.50	97.50	97.50	100.00
mix	hard	32	22	8	12.50	12.50	12.50	37.50	62.50	100.00	100.00	100.00	100.00
suzuki	easy	20	12	15	0.00	0.00	13.33	60.00	100.00	100.00	100.00	100.00	100.00
suzuki	medium	20	12	20	5.00	5.00	5.00	20.00	70.00	95.00	100.00	100.00	100.00
suzuki	hard	20	12	9	0.00	0.00	0.00	11.11	11.11	33.33	66.67	66.67	88.89
All				313	26.52	26.52	35.46	60.06	78.27	87.86	93.93	94.89	99.04

Table 10. Grading of Instances

more challenging, as long as it remains *well-posed*, i.e. keeps a unique solution. Similar to [18], we experimented with a greedy method which removes hints one by one, until multiple solutions appear. We consider two variants, in the *complete* reduction we remove the hint completely from the problem, in the *partial* reduction we keep the *alldifferent* condition and only remove the *sum* constraint. Table 11 shows the result for a subset of problem instances. We can see that for complete reduction we can remove between 3 and 16 percent of all hints without losing uniqueness, while for partial removal this increase to 6 to 37 percent.

Set	X	Y	K	Complete			Partial		
				Min	Avg	Max	Min	Avg	Max
giants	32	22	10	5.51	6.86	8.40	11.42	14.26	20.44
jnp	9	9	8	6.25	11.02	16.67	6.25	14.44	17.86
jnp	10	10	8	5.56	10.38	15.79	8.33	12.10	15.79
jnp	12	12	8	7.41	10.18	16.07	10.00	13.39	18.52
kakuro	10	14	39	3.85	10.48	15.52	12.07	21.64	36.21
kakuro	16	16	44	3.41	8.48	12.50	9.76	17.68	25.00
mix	12	12	12	4.17	9.85	14.29	13.46	21.06	37.50
mix	16	16	70	4.26	9.35	13.21	11.36	17.58	26.19
suzuki	20	12	44	6.38	10.39	13.46	11.96	18.22	26.53
All			243	3.41	9.60	16.67	6.25	17.98	37.50

Table 11. Reduction Summary

The reduced problems are significantly more difficult than the original. Table 12 shows the results for the combined model on the *partial* reduction. Note that now only a quarter of the problems are solved at setup, while even shaving is not sufficient to solve all reduced problems. Results for the *complete* reduction are slightly better, but comparable.

Set	X	Y	K	Setup	Shave	Total	Avg Time	Max Time	Avg Back	Max Back
giants	32	22	10	0.00	60.00	100.00	15.98	49.85	4.40	34
jnp	9	9	8	62.50	100.00	100.00	0.02	0.06	0.00	0
jnp	10	10	8	50.00	100.00	100.00	0.03	0.07	0.00	0
jnp	12	12	8	12.50	100.00	100.00	0.16	0.48	0.00	0
kakuro	10	14	39	41.03	100.00	100.00	0.04	0.18	0.00	0
kakuro	16	16	44	13.64	90.91	100.00	0.47	6.37	1.80	65
mix	12	12	12	41.67	100.00	100.00	0.02	0.04	0.00	0
mix	16	16	70	21.43	98.57	100.00	0.94	37.02	0.00	0
suzuki	20	12	44	22.73	100.00	100.00	0.12	0.52	0.00	0
All			243	25.51	96.30	100.00	1.05	49.85	0.51	65

Table 12. Partial Reduction Results

11 Summary

In this paper we have considered models for the Kakuro logic puzzle. A finite domain constraint model with a GAC *alldifferent-sum* constraint solves nearly all considered instances just by constraint propagation, without requiring shaving or search. Adding some redundant constraints, all examples (up to 124x90 size) are solved without search or shaving in less than 3 seconds (average 70ms). This compares favourably to (naive) models using MIP and SAT techniques, and is comparable to efficient C++ based solutions. We also considered a grading scheme to estimate the difficulty of a puzzle instance for a human, and showed that a significant number of hints in the puzzles can be removed without losing the uniqueness of the solution, generating more challenging puzzles.

References

1. Carlos Ansótegui, Ramón Béjar, Cèsar Fernández, Carla P. Gomes, and Carles Mateu. The impact of balancing on problem hardness in a highly structured domain. In *AAAI*. AAAI Press, 2006.
2. N. Beldiceanu, M. Carlsson, and J.X. Rampon. Global constraint catalog. Technical Report T2005:08, SICS, May 2005.
3. M. Carlsson. Kakuro model with gcc constraint, 2007. Personal communication.
4. Niklas En and Niklas Srensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

5. Niklas En and Niklas Srensson. Translating Pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:1–26, 2006.
6. A. Immanuel. *Japanese Number Puzzles*. Carlton Books, 2006.
7. S. K. Jones, P. A. Roach, and S. Perkins. Construction of heuristics for a search-based approach to solving SuDoku. In *Research and Development in Intelligent Systems XXIV: Proceedings of AI-2007, the Twenty-seventh SGAI International Conference on Artificial Intelligence*. Springer-Verlag, 2007.
8. Robin Lougee-Heimer. The common optimization interface for operations research. *IBM Journal of Research and Development*, 47(1):57–66, January 2003.
9. I. Lynce and J. Ouaknine. Sudoku as a SAT problem. In *9th International Symposium on Artificial Intelligence and Mathematics*, January 2006.
10. Paul Martin and David B. Shmoys. A new approach to computing optimal schedules for the job-shop scheduling problem. In William H. Cunningham, S. Thomas McCormick, and Maurice Queyranne, editors, *IPCO*, volume 1084 of *Lecture Notes in Computer Science*, pages 389–403. Springer, 1996.
11. Gilles Pesant. A regular language membership constraint for finite sequences of variables. In Mark Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 482–495. Springer, 2004.
12. Thierry Le Provost and Mark Wallace. Generalised constraint propagation over the CLP scheme. *Journal of Logic Programming*, 16(3):319–360, 1993.
13. Christopher G. Reeson, Kai-Chen Huang, Kenneth M. Bayer, and Berthe Y. Choueiry. An interactive constraint-based approach to Sudoku. In *AAAI*, pages 1976–1977. AAAI Press, 2007.
14. Jean-Charles Régin. Arc consistency for global cardinality constraints with costs. In Joxan Jaffar, editor, *CP*, volume 1713 of *Lecture Notes in Computer Science*, pages 390–404. Springer, 1999.
15. Joachim Schimpf and Kish Shen. ECLiPSe by example, 2007. Tutorial at CP-2007.
16. Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *CoRR*, abs/cs/0611009, 2006.
17. Kish Shen and Joachim Schimpf. Eplex: Harnessing mathematical programming solvers for constraint logic programming. In Peter van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 622–636. Springer, 2005.
18. H. Simonis. Sudoku as a constraint problem. In B. Hnich, P. Prosser, and B. Smith, editors, *Proceedings of the 4th International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 13–27, September 2005.
19. Y. Suzuki. *The Giant Book of Japanese Puzzles*. Arcturus, 2006.
20. S. Thiel. *Efficient Algorithms for Constraint Propagation and for Processing Tree Descriptions*. PhD thesis, Universität des Saarlandes, 2004.
21. Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.
22. various. *Huge Sheet of Kakuro 2*. Nikoli, 2001. In Japanese.
23. various. *101 Kakuro*. Nikoli, 2003. In Japanese.
24. various. *Penpa Mix 1-4*. Nikoli, 2004. In Japanese.
25. various. *Puzzle the Giants Vol 1-6,17,18,19,20*. Nikoli, 2004. In Japanese.
26. various. Kakuro, 2007. <http://en.wikipedia.org/wiki/Kakuro>.
27. various. Nikoli web site, 2007. <http://www.nikoli.co.jp/en>.
28. M. Wallace, S. Novello, and J. Schimpf. ECLiPSe : A platform for constraint logic programming. *ICL Systems Journal*, 12(1), May 1997.