Constraint Programming and Combinatorial Optimisation in Numberjack

Emmanuel Hébrard, O'Mahony Eoin, O'Sullivan Barry

Constraint Programming and Combinatorial Optimisation in Numberjack*

Emmanuel Hebrard[1,2], Eoin O'Mahony[1], and Barry O'Sullivan[1]

[1] Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
{e.hebrard|e.omahony|b.osullivan}@4c.ucc.ie
[2] LAAS-CNRS Toulouse, France
hebrard@laas.fr

Abstract. Numberjack is a modelling package written in Python for embedding constraint programming and combinatorial optimisation into larger applications. It has been designed to seamlessly and efficiently support a number of underlying combinatorial solvers. This paper illustrates many of the features of Numberjack through the use of several combinatorial optimisation problems.

1 Introduction

We present Numberjack[3], a Python-based constraint programming system. Numberjack brings the power of combinatorial optimisation to Python programmers by supporting the specification of complex problem models and specifying how these should be solved. Numberjack provides a common API for constraint programming, mixed-integer programming and satisfiability solvers. Currently supported are: the CP solvers Mistral and Gecode; a native Python CP solver; the MIP solver SCIP; and the satisfiability solver MiniSat[4]. Users of Numberjack can write their problems once and then specify which solver should be used. Users can incorporate combinatorial optimisation capabilities into any Python application they build, with all the benefits that it brings.

2 Modelling in Numberjack

Numberjack is provided as a Python module. To use Numberjack one must import all Numberjack's classes, using the command: `from Numberjack import *`. Similarly, one needs to import the modules corresponding to the solvers that will be invoked in the program, for instance: `import Mistral` or `import Gecode`. The Numberjack module essentially provides a class `Model` whereas the solver modules provide a class `Solver`, which are built from a `Model`. The structure of a typical Numberjack program is presented in Figure 1. Notice that it is possible to use several types of solver to solve the same model by explicitly invoking the modules. To solve a model, the various methods implemented in the back-end solvers can be invoked through Python.

* Supported by Science Foundation Ireland Grant Number 05/IN/I886.

[3] Available under LGPL from `http://numberjack.ucc.ie`

[4] Mistral: `http://4c.ucc.ie/~ehebrard/Software.html`; Gecode: `http://gecode.org`; SCIP: `http://scip.zib.de/`; MiniSat: `http://minisat.se`;

```
from Numberjack import * # Import all Numberjack classes
import Gecode            # Import the Gecode solver interface
import Mistral           # Import the Mistral solver interface

model = Model()          # Declare a new model
...                      # Define the constraints and objectives

gsolver = Gecode.Solver(model)   # Declare a Gecode solver
msolver = Mistral.Solver(model)  # Declare a Mistral solver
gsolver.solve()                  # Solve the model with Gecode
msolver.solve()                  # Solve the model with Mistral
```

Fig. 1. The structure of a typical Numberjack program.

Almost every statement in Numberjack is an *expression*. Variables are expressions, and constraints are expressions on a set of sub-expressions. Variable objects are created by specifying its domain by passing a lower and an upper bound, or a set of values. One can also use floating point values for the bounds, however the result will depend on the back-end solver. MIP solvers will, by default, treat variables declared with floating point values as continuous and integer otherwise. A model is a set of expressions.

It is possible to define classes of objects to help write concise models. For instance, the objects `VarArray` and `Matrix` are syntactic sugars for one-dimensional and two-dimensional arrays of Numberjack expressions, respectively. The `Matrix` object allows us to reference the rows, the columns, and a flattened version of the matrix using `.row`, `.col` and `.flat`, respectively. The overloaded bracket (`[]`) operator tied to the Python object method `getitem`. The operator takes one argument representing the index of the object to be returned. For `VarArray` and `Matrix` objects this argument can either be a Numberjack expression or an integer. The bracket operator of the `Matrix` object returns the `VarArray` object representing the row at the given index. When the index argument is itself a Numberjack expression, the result is interpreted as an `Element` constraint. Objective functions are also expressions.

2.1 Some Example Models

Costas Array. A Costas array[5] is an arrangement of N points on a $N \times N$ checkerboard, such that each column or row contains only one point, and that all of the $N(N-1)/2$ vectors defined by these points are distinct. We model this problem in Figure 2 as follows: for each row, we introduce a variable whose value represents the column at which a point is placed in this row. To ensure that no two points share the same column, we post an `AllDiff` constraint on the rows (Line 4). To each value $y \in [1..N-2]$, we can map a set of vectors whose vertical displacements are equal, i.e., the vectors defined by the points $(row[i], i)$ and $(row[i+y], i+y)$. To ensure that these vectors are distinct, we use another `AllDiff` constraint.

Golomb Ruler. In the Golomb ruler problem the goal is to minimise the position of the last mark on a ruler such that the distance between each pair of marks is different. The Numberjack model is shown in Figure 3.

[5] `http://mathworld.wolfram.com/CostasArray.html`

```
N   = 10
row = [Variable(1,N) for i in range(N)]

model =  Model()
model += AllDiff(row)
for y in range(N-2):
   model += AllDiff([row[i] - row[i+y+1] for i in range(N-y-1)])

solver = Mistral.Solver(model)
solver.solve()
```

Fig. 2. A Numberjack model for a 10×10 instance of the Costas array problem.

```
M = [Variable(1,rulerSize) for i in range(N)]

model =  Model()
model += AllDiff([M[i]-M[j] for i in range(1,N) for j in range(i)])

model += Minimise(marks[nbMarks-1])  # The objective function
```

Fig. 3. A Numberjack model for the Golomb ruler problem.

Magic Square. In this problem one wants every number between 1 and N^2 to be placed in an $N \times N$ matrix such that every row, column and diagonal sum to the same number. A model for that problem making use of the `Matrix` class is presented in Figure 4.

```
N = 10
sum_val = N*(N*N+1)/2
square  = Matrix(N,N,1,N*N)

model = Model(

   # The values in each cell must be distinct
   AllDiff(square.flat),

   # Each row and column must add to sum_val
   [Sum(row) == sum_val for row in square.row],
   [Sum(col) == sum_val for col in square.col],

   # Each diagonal must add to sum_val
   Sum([square[a][a] for a in range(N)]) == sum_val,
   Sum([square[a][N-a-1] for a in range(N)]) == sum_val )
```

Fig. 4. A Numberjack model for the Magic Square problem.

Quasigroups. A quasigroup is $m \times m$ multiplication defined by a matrix which from a Latin square, i.e. every element occurs once in every row and column. The result of the product $a * b$ corresponds to the element at row a and column b of the matrix. Figure 5 presents a model for the problem in which for all a, b we have: $((b * a) * b) * b = a$.

2.2 Extending Numberjack

Numberjack provides a facility to add custom constraints. Consider the following optical network monitoring problem taken from [1]. An optical network consists of nodes

```
N = 8
x = Matrix(N,N,N)

model = Model(

   # The rows and columns form a Latin square
   [AllDiff(row) for row in x.row],
   [AllDiff(col) for col in x.col],

   # Enforce the QG5 Property
   [x[ x[  x[b][a] ][ b ] ][b] == a for a in range(N) for b in range(N)] )
```

Fig. 5. A Numberjack model for the Quasigroup Existence problem.

and fibre channels. When a node fails in the network, all lightpaths passing through that node are affected. Monitors attached to the nodes present in the affected lightpaths trigger alarms. Hence, a single fault will generate multiple alarms. By placing monitors in the right way, we can minimize the number of alarms generated for a fault while keeping the fault-detection coverage maximum. In the problem we model below, we add the additional constraint that for any node failure that might occur, it triggers a unique set of alarms. This problem requires that each combination of monitor alarms is unique for each node fault. This requires that every pair of vectors of variables differ on at least one element. This can be specified in Numberjack by introducing a `HammingDistance` constraint. The Numberjack model for this problem is presented as Figure 6.

```
class HammingDistance(Expression):

 def __init__(self, row1, row2):
   Expression.__init__(self, "HammingDistance")
     self.set_children(row1+row2)
     self.rows = [row1, row2]

 def decompose(self):
   return [Sum([(var1 != var2) for var1, var2 in zip(self.rows[0],self.rows[1])])]

Nodes = 6        # We consider a graph with 6 nodes
Monitors = 10    # Faults on the nodes trigger 10 monitors

alarm_matrix = [ # Each vector specifies the monitors triggered by each node
   [1, 2, 3,                  10], [                 7          ],
   [               6, 7,        ], [            5, 6, 7,        ],
   [   2, 3, 4,          8,    10], [      3, 4,          8, 9, 10]  ]

monitors_on     = VarArray(Monitors)        # The decision variables
being_monitored = Matrix(Nodes, Monitors)

model = Model()                             # Specify the model...
model.add( Minimise(Sum(monitors_on)) )
model.add( [ monitor == ( Sum(col) >= 1 ) for col, monitor in
                        zip(being_monitored.col, monitors_on) ])
model.add( [ Sum(row) > 0 for row in being_monitored] )
model.add([HammingDistance(x1,x2) > 0 for x1, x2 in pair_of(being_monitored)])
for monitored_row, possible_monitor_row in zip(being_monitored, alarm_matrix):
   model.add([monitored_row[idx - 1] == 0 for idx in
              [x for x in range(Monitors) if x not in possible_monitor_row]])
```

Fig. 6. A Numberjack model for the optical network monitoring problem [1].

3 Experiments

Experiments ran on an Intel Xeon 2.66GHz machine with 12GB of ram on Fedora 9.

Experiment 1: Overhead due to Numberjack. We first assess the overhead of using a solver within Numberjack. We ran three back-end solvers, Mistral, MiniSat and SCIP on three arythmetic puzzles (Magic Square, Costas Array and Golomb Ruler). For each run, we used a profiler to separate the time spent executing Python code from the time spent executing code from the back-end solver. We report the results in Table 1. For every problem we report results averaged across 7 instances[6] of various size and 10 randomized runs each. The time spent executing the Python code is very modest, and of course independent of the hardness of the instance.

Table 1. Solver Time vs Python Time (Arithmetic puzzles).

Instance	Mistral Time (s)		MiniSat Time (s)		SCIP Time (s)	
	Solver	Python	Solver	Python	Solver	Python
`Magic-Square` (3 to 9)	0.0205	0.0101	59.7130	0.0116	35.85	0.0107
`Costas-Array` (6 to 12)	0.0105	0.0098	0.0666	0.0095	78.2492	0.0134
`Golomb-Ruler` (3 to 9)	0.5272	0.0056	56.0008	0.0055	118.1979	0.0076

Experiment: Comparison of Back-end Solvers. It is well known in the fields of Constraint Programming and Mixed Integer Programming that the areas have different strengths and weaknesses. For example, the CP and SAT solvers were much more efficient than the MIP solver for the arythmetic puzzles used in the first set of experiments (See Table 1). However, of course, the situation can be completely reversed on problems more suited to mathematical programming. We ran Numberjack on the Warehouse allocation problem (P34 of the CSPLib). This problem is easily solved using the Mixed Integer Solver SCIP as back end (1.86 seconds and 4.8 nodes in average over the 5 instances on the CSPLib) whilst Mistral ran over a time limit of one hour, staying well over the optimal allocation and exploring several million nodes.

4 Conclusion

Numberjack is a Python-based constraint programming system. It brings the power of combinatorial optimisation to Python programmers by supporting the specification of complex models and specifying how these should be solved. We presented the features of Numberjack through the use of several combinatorial problems.

References

1. Puspendu Nayek, Sayan Pal, Buddhadev Choudhury, Amitava Mukherjee, Debashis Saha, and Mita Nasipuri. Optimal monitor placement scheme for single fault detection in optical network. In *Proceedings of 2005 7th International Conference*, pages 433–436 Vol. 1, 2005.

[6] The results of SCIP on the 3 hardest Magic Square instances are not taken into account since the cutoff of 1000 seconds was reached.