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. Python benefits from a large and active programming community. 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. Currently, Numberjack supports three constraint programming solvers, one MIP solver, and one satisfiability solver – all available as open-source software. This paper illustrates many of the features of Numberjack through the use of several combinatorial optimisation problems. We also demonstrate a cloud-based configurator built with Numberjack, using services provided by Google to support a user-interface and back-end reasoning capabilities.

1 Introduction

Constraint programming provides powerful support for decision-making; it is able to search quickly through an enormous space of choices, and infer the implications of those choices. Advances in constraint programming technology have been implemented in a variety of software toolkits, modelling languages with underlying solvers, and systems. For example, both OPL [7] and Minizinc [10] provide rich modelling functionalities in a high level language especially designed for specifying combinatorial optimisation problems. Other systems, such as CHIP [4], Prolog III [3], Comet [13], Gecode[3], IBM ILOG CP[4], and CPInside [5], provide general programming capabilities. Many systems also exist for combinatorial optimisation using mixed-integer programming, e.g. IBM ILOG CPLEX[5] and SCIP [1]. While constraint programming already has wide commercial application, much remains to be done to fully explore and exploit the technology. A major concern at present is the ease-of-use of the technology [11].

In this paper we present Numberjack, 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 constraint programming, mixed-integer

[3] http://www.gecode.org/
[4] http://www-01.ibm.com/software/integration/optimization/cp1/
[5] http://www-01.ibm.com/software/integration/optimization/cplex/

programming and satisfiability solver back-ends. At present Numberjack supports the CP solver Mistral, a native python solver, the MIP solver SCIP, and the satisfiability solver MiniSat. Users of Numberjack can write their problems once and then specify which solver should be used. Since Numberjack is a Python-based system, users also incorporate combinatorial optimisation capabilities into Python programs, and all the benefits that brings. For example, we will describe how constraint-based applications can be easily built for the web using Google's AppEngine infrastructure.

The remainder of this paper is organised as follows. In Section 2 we present the basics of modeling combinatorial problems using Numberjack. In Section 3 we show how complex search strategies can be written in Numberjack. In Section 4 we present how a cloud-based application can be easily built and deployed using Numberjack. A detailed empirical evaluation of the performance of Numberjack is presented in Section 5. Concluding remarks are made in Section 6.

2 Modelling in Numberjack

The general Numberjack framework is comparable to what constraint programmers are used to from other toolkits. One can define variables, constraints and models, call solvers on the models and query them for solutions and statistics. First, as for every *module* in Python, one needs to import all Numberjack's classes, through the following 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 SCIP`.

The Numberjack module essentially provides a class `Model` whereas the solver modules provide a class `Solver`, which are built from a `Model`. In the case of CP solvers, the model is straightforwardly converted into the solver's internal structure, although we will see in Section 2.4 that one can define decomposition for constraints that are not supported by a given solver. On the other hand, for SAT and MIP solvers such as MiniSat and SCIP, the process of converting Numberjack's code into the solver's language may not be trivial. It is beyond the scope of this paper to detail CSP to SAT or CSP to MIP encoding. However, we use the same principle in both cases. We choose from two types of encoding for variable domains depending on the requirements of the constraints. For SAT, we use either the *order* encoding [12] for linear constraints or the *direct* encoding otherwise. For MIP, in the same situations respectively, we use either a single integer variable to represent the bounds of the encoded variable, or a set of Boolean variables each standing for a different value. When a variable, in the Numberjack model, is constrained simultaneously by linear and non-linear constraints, both representations are channeled for the underlying solver. To encode constraints, we use standard formulation, from for example [8] and [12].

The structure of a typical Numberjack program is presented in Figure 1. Notice that it is possible to use several types of solver by explicitly invoking the modules. To solve the model, the various methods implemented in the back-end solvers can be invoked through python. This part is detailed in Section 3.

Note that the examples given in subsequent sections use some python functions and syntax which may need to be defined[6]. The Python function `(zip)` takes a variable number of lists as arguments and returns a list of tuples the first tuple containing the first element of each list and so on. The other syntactic sugar used is Python's list comprehension capability which takes the following syntax `[function(x) for expression in generator (if predicate)]`. The predicate is optional, the function can be the identity function or any other operation. The generator can be a list or any object that can be iterated on.

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

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

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

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

2.1 Expressions

Almost every statement in Numberjack is an *expression*. Variables are expressions, and constraints are expressions on a set of sub-expressions. Variables objects are created by passing a lower and an upper bound, or a set of values:

```
x = Variable(1, 1000)
y = Variable([3,5,7,9])
z = Variable(2.0, 7.0)
```

Notice that 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 in fact a set of expressions that can be passed as arguments at creation:

```
model = Model( x < y, y < z )
```

or with either the `add` method, or the operator += as follows:

```
model = Model()
model.add(x < y)
model += (y < z)
```

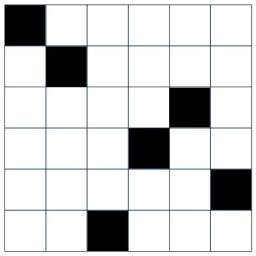
[6] The reader is referred to Python's documentation for more details.

Since constraints are themselves expressions, it is possible to state nested constraints in an intuitive way:

```
model += (x+y > z)
```

2.2 Two Examples

Costas Array. A Costas array[7] 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 this points are distinct. An example is presented in Figure 2: the array is on the left.



```
1  2  5  4  6  3
-1 -3  1 -2  3
-4 -2 -1  1
-3 -4  2
-5 -1
-2
```

Fig. 2. Solution to a 6x6 Costas array on left and Costas triangle on the right.

We model this problem in Figure 3 as follows: for each row, we introduce a variable whose value represents the column at which a point for this row. In order to ensure that there is no two points share the same column, we post and `AllDiff` constraint on the rows (Line 04). To each value $y \in [1..N-1]$, 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 to ensure that corresponding displacements on the x-axis are distinct. Notice that `row[i] - row[i+y+1]` return a Numberjack's expression equal to the difference between `row[i]` and `row[i+y+1]`. The triangle formed by these differences is shown on the right of Figure 2.

Golomb Ruler. Objective functions are also expressions and follow the same syntax as constraints. For instance, in the Golomb ruler problem the goal is to minimise the size - that is, the value of the last mark - of a ruler such that the distance between each pair of marks is different. This objective function can be added to the model using the keyword `Minimise`, as shown in Figure 4.

[7] `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. 3. A Numberjack model for 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])  # Declare the objective function
```

Fig. 4. A Numberjack model for the Golomb ruler problem.

2.3 Language Features to Support Concise Models

Since Python is an object oriented language, it is possible to define classes of objects to help modelling problems. For instance, the objects `VarArray` and `Matrix` are such syntactic sugars used for one-dimensional and two-dimensional arrays of Numberjack expressions, respectively. An object `VarArray` is very similar to a Python type `list` containing `Variable` objects. It is indeed possible to use Python lists as shown in Figure 3 and 4. However, a number of operator are overridden for `VarArray`'s which makes them more adapted to constraint programming. For instance, if we replace the definition of `row` with `row = VarArray(N,1,N)`, then we can print the solution using `print row`. This is due to an overridden to string method in `VarArray`. These constructs are necessary to post *Element* constraints or *LexOrder* constraints by using, respectively, the operators `[]` and $\leq$.

The `Matrix` object is slightly more complex. It allows us to reference the rows, the columns, and a flattened version of the matrix using `.row`, `.col` and `.flat`, respectively. For instance, consider the well known Magic square problem, where one wants every number between 1 and N^2 to be placed in a $N \times N$ matrix such that every row, column and diagonal add up to the same number. A model for that problem making use of the `Matrix` class is presented in Figure 5.

Another example of overridden operator is the bracket (`[]`) operator tied to the python object method `getitem`. The operator needs one argument representing the index of the object to be returned. For `VarArray` and `Matrix` objects this argument can be 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.

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

square = Matrix(N,N,1,N*N)
model = Model(
   AllDiff(square.flat),                                  # Distinct cells

   [Sum(row) == sum_val for row in square.row],           # Rows
   [Sum(col) == sum_val for col in square.col],           # Columns

   Sum([square[a][a] for a in range(N)]) == sum_val,      # Down Diagonal
   Sum([square[a][N-a-1] for a in range(N)]) == sum_val # Up Diagonal
 )
```

Fig. 5. A Numberjack model for the Magic Square problem.

When the index argument is itself a Numberjack expression, the result is interpreted as an `Element` constraint. For instance, consider the Quasigroup existence problem, where we want to find a quasigroup with some properties. This is problem `prob003` in the CSPLIB. A quasigroup is $m \times m$ multiplication defined by a matrix which from a Latin square, that is, every each element occurs once in every row and column. The result of the product $a * b$ thus corresponds to the element at row a and column b of the matrix. Let consider the quasigroups denoted "QG5" where for all a, b we have: $((b * a) * b) * b = a$. A model is presented in Figure 6.

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

model = Model(
   [AllDiff(row) for row in x.row], # latin square (rows)
   [AllDiff(col) for col in x.col], # latin square (columns)
   # property QG5
   [x[ x[  x[b][a] ][ b ] ][b] == a for a in range(N) for b in range(N)]
)
```

Fig. 6. A Numberjack model for the Quasigroup Existance problem.

A number of such helpers are implemented in the kernel language, for instance (`task`, `resource`, `input`, `pair_of`, etc.) However, since Numberjack is an open API in Python it is possible, for the user to directly enrich the language with such constructs.

2.4 Extending Numberjack

Numberjack provides a facility to add custom constraints. Suppose one of Numberjack's solver back-ends has a new global constraint that is needed for a problem, adding this new constraint to Numberjack is easily achieved.

Consider the following optical network monitoring problem taken from [9]. An optical network consists of nodes and fiber channels: the nodes model transmitters, receivers, add-drop multiplexers, etc., and they are connected by edges representing the

```
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])]) ]
```

Fig. 7. Adding the `HammingDistance` constraint to Numberjack.

optical fibre topology of the network. Fiber channels are different wavelengths, established lightpaths on the network. 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. We want to 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 an `AllDifferent` constraint across vectors of variables. This can be specified in Numberjack by introducing a `HammingDistance` constraint as shown in Figure 7. We introduce a class that extends Numberjack's `Expression` class. If this constraint is present in an underlying solver, Numberjack will use the constraint, otherwise Numberjack will decompose the constraint.

```
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]])

import Mistral                  # Import the Mistral interface
solver = Mistral.Solver(model)  # Solve the model with Mistral
if solver.solve():              # If there is a solution:
   print monitors_on           #    - print which monitors we use
else: print 'No solution'
```

Fig. 8. A Numberjack model for the optical network monitoring problem [9].

Using this feature of Numberjack one can implement custom constraints in terms of any constraints available in the underlying solvers, providing significant scope for implementing interesting combinations of constraints. A complete implementation of the Numberjack model of the optical network monitoring problem is presented as Figure 8.

3 Advanced Search Strategies in Numberjack

Constraint programming involves solving declaratively specified models. It is, therefore, often difficult to write search procedures that deviate too much from the usual setting: state the constraints; tune some parameters; query a solver. We illustrate Numberjack's capabilities through an example of a program for jobshop scheduling.

An $n \times m$ job shop problem involves a set of *tasks* $\mathcal{T} = \{t_{ij}\}_{1 \leq i \leq n,\ 1 \leq j \leq m}$ partitioned into n *jobs* that need to be scheduled on a set of m *machines*. Each task is associated to a duration, or processing time p_{ij}. A *schedule* is a mapping of tasks to time points consistent with *sequencing* constraints to ensure that within each job, the tasks have to run in the predefined order of their indices and *resource* constraints to ensure that no two tasks run simultaneously on any given machine. We consider the standard objective function defined as the minimisation of the *makespan* C_{max}, that is, the total duration to run all tasks.

We implement a search strategy described in [6] that uses two components. The first component tries to quickly narrow the gap between the lower and upper bounds on the makespan by performing a dichotomic search. It is very important since the time horizon can be arbitrarily large, and therefore a pure Branch & Bound procedure, by progressing by incremental steps, can be very slow. Then, when the gap is sufficiently small, a classical Branch & Bound procedure takes over. Implementing these components in Numberjack is simple.

First, we implement a class `JSP` to represent the instance. The actual code for this class is omitted here, however we assume that the class has three fields:

- `self.job`, a 2-D matrix storing the duration of each task t_{ij}.
- `self.machine`, a list of lists, one per machine m_j storing the pairs (i, j) such that task t_{ij} requires m_j.
- `self.get_machine`, a 2-D matrix storing the machine index of each task t_{ij}.

The code to get initial lower and upper bounds is given in Figure 9. The lower bound is simply initialised as the maximal total duration of a set of tasks corresponding to either a job or a machine. The initial upper bound is obtained by scheduling every task in turn to the earliest possible starting time, starting with t_{11} up to t_{nm}.

Using this data, we create a class `JSP_Model` that inherits from Numberjack's `Model` class (Figure 10). The only difference is that we store the sequence of tasks-ordering variables to branch on and, in a separate array, the variables standing for the start time of the tasks. The method `set_makespan(C_max)` simply resets the due date of each task to be consistent with a new makespan C_{max}.

We then declare a function to solve this model (Figure 11), by calling Mistral and setting some parameters up. The solver object is returned since it contains all the needed statistics once the problem has been solved.

```
class JSP:
   [...]                 # parse a data file and initialise the fields

   def lower_bound(self): # longest total task's duration for a job or a machine
       longest_job = max([sum(job) for job in self.job])
       longest_machine = max([sum([self.job[i][j] for (i,j) in mac])
                                                  for mac in self.machine])
       return max([longest_job, longest_machine])

   def upper_bound(self): # schedule the tasks ``as they come''
       M_job = [0]*self.nJobs
       M_machine = [0]*self.nMachines
       for i in range(self.nMachines):
           for j in range(self.nJobs):
               start_time = max(M_job[j], M_machine[self.m[j][i]])
               M_job[j] = start_time+self.job[j][i]
               M_machine[self.m[j][i]] = start_time+self.job[j][i]
       return max(max(M_job), max(M_machine))
```

Fig. 9. Implementing the bounds on makespan in Numberjack.

```
class JSP_Model(Model):

   def __init__(self,jsp,C_max=1000000):
       Model.__init__(self)

       Tasks = Matrix([[Task(C_max, p) for p in job] for job in jsp.job])
       Machines = [UnaryResource([Tasks[m] for m in machine]) for machine in jsp.machine]
       for task in Tasks.row:
           self += [task[i] < task[i+1] for i in range(jsp.nMachines-1)]
       self += Machines
       self.sequence = [d for machine in Machines for d in machine]
       self.tasks = Tasks.flat

   def set_makespan(self,C_max):  # used to change the target makespan
       for task in self.tasks: task.reset(C_max)
```

Fig. 10. Creating the jobshop model.

```
def solve(model, best_solution):
   solver = Solver(model, model.sequence)
   solver.setHeuristic('Scheduling', 'Promise')
   if best_solution != None: solver.guide(model.sequence, best_solution)
   solver.setRestartNogood()
   solver.setTimeLimit(300)
   solver.solveAndRestart(GEOMETRIC, 256, 1.3)
   return solver
```

Fig. 11. Specifying how the JSP model should be solved in Numberjack.

```
def dichotomic_search(model, LB, UB):
   lb = LB                                          # temporary lower bound
   best_solution = None

   while lb < UB:
       C_max = int((lb + UB) / 2)                   # dichotomic pivot
       model.set_makespan(C_max)
       outcome = solve(model, best_solution, 7000)
       if outcome.is_sat():
           UB = max([task.get_value() + task.duration for task in model.tasks])
           best_solution = [x.get_value() for x in model.sequence]
       else:
           lb = C_max+1
           if outcome.is_unsat(): LB = C_max+1  # the lower bound is proven

   return (LB, UB, best_solution)

def branch_and_bound(model, LB, UB, best_solution):
   model.set_makespan(UB-1)
   C_max = Variable(LB, UB-1)
   for task in model.tasks: model.add(task < C_max)    # link the objective
   model.add( Minimise(C_max) )

   outcome = solve(model, best_solution, 1000000)
   if outcome.is_sat(): UB = C_max.get_value()
   if outcome.is_opt() or outcome.is_unsat(): LB = UB  # proven optimal

   return (LB, UB, best_solution)

def schedule(jsp):
   model = JSP_Model(jsp)
   (LB, UB, best) = dichotomic_search(model, jsp.lower_bound(), jsp.upper_bound())
   if LB < UB: (LB, UB, best) = branch_and_bound(model, LB, UB, best)
   return (LB, UB)
```

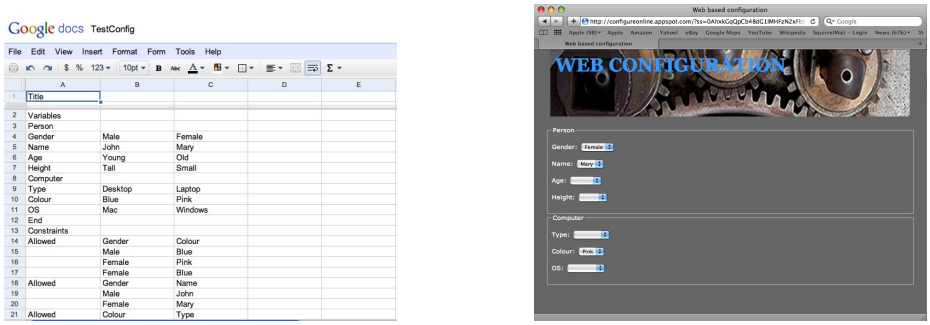
Fig. 12. Implementing the dichotomic search and Branch & Bound in Numberjack, which are then used in the scheduler.

Finally, the two components of our solving strategy – dichotomic search and Branch & Bound – can be implemented (Figure 12). The dichotomic search proceeds by solving the decision problem of the existence of a schedule for a makespan equal to the mean of the current lower and upper bounds. The bounds are updated according to the result. Observe that since a time cutoff is imposed on the search, we need to distinguish the real lower bound (`LB`) from the one used for the dichotomic phase (`lb`). To implement the Branch & Bound procedure we add a variable `C_max` standing for the objective value, link it to the model, and post an objective function. Then, the overall method to solve an instance of the job-shop scheduling problem can be stated very simply, as shown in that same figure.

4 Deploying Numberjack Applications on the Cloud

One of the strengths of Numberjack, due to Python's prowess as a web language is that it can be easily deployed in online applications. Google provides a solution for easy deployment of web applications with their Google AppSpot framework[8]. To demonstrate the ease with which Numberjack can be deployed in the cloud we designed

[8] `http://appspot.com`



(a) Specifying the configuration problem in a Google Spreadsheet.

(b) The GUI of the online configuration application.

Fig. 13. An example cloud-based application built using Numberjack.

an online product configurator (Figure 13). The configuration problem is defined in a Google Spreadsheet (Figure 13(a)) The spreadsheet is split into two separate parts. Firstly, the options are specified. The first column contains the name of each variable and the remaining contain the values in their domain. Secondly, the spreadsheet contains the constraints of the problem, specified in terms of allowed tuples. The online application parses the spreadsheet using Google's Documents API, available in Python, and produces a Numberjack model of the configuration problem along with a web page presenting the options to the user (Figure 13(a)). As the user makes choices constraint propagation is performed on the server side. The solver used is the Python solver included in Numberjack. Any spreadsheet can be used to create an online configurator by passing the URL of the Google spreadsheet to the web application[9].

This application highlights the ease of deploying web applications powered by constraint programming using Numberjack. Applications like the online configurator can be designed, implemented and deployed within hours.

5 Experiments

In this section we assess the overhead of using a solver within Numberjack. The overhead is twofold. The Python code is interpreted and therefore generally slower to run than compiled code. Moreover, the interpreted nature of Python tends to add a slight overhead to execution, even of the compiled object code of the back-end solvers. For instance, hooks are provided to handle cleanly interruptions.

We also use these experiments to showcase the benefit of Numberjack for easily comparing different solvers on the same problem. All experiments reported in this paper ran on an Intel Xeon 2.66GHz machine with 12GB of ram on Fedora 9.

[9] Available on `http://configureonline.appspot.com`.

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)	0.0004	0.0046	0.0073	0.0048	0.0230	0.0087
`Magic-Square` (4)	0.0009	0.0055	0.0799	0.0056	6.0881	0.0096
`Magic-Square` (5)	0.0040	0.0070	0.9774	0.0074	28.9815	0.0111
`Magic-Square` (6)	0.0119	0.0090	10.9969	0.0112	108.3293	0.0134
`Magic-Square` (7)	0.0137	0.0119	24.7692	0.0148	-	-
`Magic-Square` (8)	0.0318	0.0149	171.2539	0.0179	-	-
`Magic-Square` (9)	0.0810	0.0175	209.9064	0.0200	-	-
`Costas-Array` (6)	0.0004	0.0046	0.0030	0.0043	4.0855	0.0084
`Costas-Array` (7)	0.0006	0.0056	0.0054	0.0055	8.2916	0.0096
`Costas-Array` (8)	0.0008	0.0071	0.0101	0.0070	11.4558	0.0111
`Costas-Array` (9)	0.0020	0.0094	0.0229	0.0097	25.9925	0.0136
`Costas-Array` (10)	0.0058	0.0140	0.0368	0.0113	49.1780	0.0152
`Costas-Array` (11)	0.0064	0.0127	0.0863	0.0129	249.7409	0.0167
`Costas-Array` (12)	0.0581	0.0155	0.3022	0.0160	199.0003	0.0196
`Golomb-Ruler` (3)	0.0002	0.0025	0.0003	0.0020	0.0141	0.0067
`Golomb-Ruler` (4)	0.0003	0.0032	0.0016	0.0028	0.0216	0.0076
`Golomb-Ruler` (5)	0.0006	0.0038	0.0093	0.0038	0.9819	0.0081
`Golomb-Ruler` (6)	0.0028	0.0048	0.0848	0.0051	2.9578	0.0089
`Golomb-Ruler` (7)	0.0354	0.0072	1.0669	0.0064	18.5908	0.0105
`Golomb-Ruler` (8)	0.2491	0.0078	15.7713	0.0081	804.8193	0.0119
`Golomb-Ruler` (9)	3.4026	0.0105	375.0718	0.0106	-	-

5.1 Experiment 1: Overhead of using Numberjack

We first try to characterise the overhead of using Python within a "standard" use of Numberjack, that is, stating the problem, passing it to the solver, and setting the parameters up. 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.

The default search strategy was used for MiniSat and SCIP. On the other hand Mistral does not really implement a default search strategy, therefore the following parameter setup was used for these three benchmarks:

```
solver.setHeuristic('DomainOverWDegree', 'RandomSplit', 2)
solver.solveAndRestart(GEOMETRIC, 256, 1.3)
```

The first line sets the variable ordering heuristic to a random choice between the two variables minimising the *domain/weighted degree* [2] criterion, and the branching to a domain split around a random pivot. The second line sets the restart policy to an initial failures limit of 256 increasing geometrically with a factor 1.3. We report the results in Table 1. The time spent executing the Python code is very modest, and of course independent of the hardness of the instance. This is not surprising since modelling represents only a small fraction of the computation cost.

However, one can do much more than simply stating a problem and calling a solver in Numberjack. We therefore run the same type of experiment on the algorithm for Jobshop scheduling introduced in Section 3. Here, the amount of work done within Python is not negligible. The data file is parsed, and a model is created accordingly, initial upper and lower bounds are computed, and finally the dichotomic search is controlled

Table 2. Solver Time vs Python Time (Job-shop Scheduling).

Instance	stats			Mistral	Numberjack-Mistral		
	optimal	avg	Nodes	Time (s)	Time (s)	Python (s)	Overhead
Taillard (01)	1231	1231	93048	13.34	14.46	1.72	+8.4%
Taillard (02)	1244	1244	336445	94.06	99.83	2.06	+6.1%
Taillard (03)	1218	1218	495076	101.28	106.96	2.09	+5.6%
Taillard (04)	1175	1175	206267	31.77	33.45	1.79	+5.3%
Taillard (05)	1224	1228	1153076	210.67	228.84	2.10	+8.6%
Taillard (06)	1238	1256	1260659	214.05	228.41	2.03	+6.7%
Taillard (07)	1227	1227	659347	160.78	175.00	2.07	+8.8%
Taillard (08)	1217	1217	509891	119.22	128.74	1.95	+8.0%
Taillard (09)	1274	1274	1109512	236.32	257.55	1.99	+9.0%
Taillard (10)	1241	1254	1121461	167.33	180.97	2.03	+8.1%

from within Python. We used the method described above to compute the time spent executing the Python code, shown in penultimate column of Table 2. Moreover, we also try to capture the overhead of running Mistral as a dynamic library. In order to do so, we implemented the same exact algorithm in C++, and compare the cpu times. Of course some of the difference is due to the Python code. However, this overhead is constant, whilst the overhead due to the dynamic linking increase linearly with the computational effort.

In the first three columns of Table 2, we report, respectively, the optimal objective value (makespan) for the instance, the average found by our algorithm, and the total number of nodes explored. Moreover, we report the cpu-time required by Mistral (stand-alone), the cpu-time required by Mistral (within Numberjack), the cpu-time spent executing the Python code, and finally the total overhead between the two versions of mistral in percentage:

$$\text{Overhead} = \frac{100 * (\text{Time}(\text{Mistral}) - \text{Time}(\text{Numberjack} - \text{Mistral}))}{\text{Time}(\text{Mistral})}$$

All results are averaged across 20 runs with different random seeds.

5.2 Experiment 2: Comparing Numberjack's Solver Back-ends

It is well known in the fields of Constraint Programming and Mixed Integer Programming that the areas have different strengths and weaknesses. Numberjack can exploit this by allowing a change of solver without the hassle of changing the model. A tool like Numberjack can therefore be extremely useful for quick prototyping of algorithms and/or analysis of problems. 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 used the same parameter setup as for the arythmetic puzzles for both solvers.

Figure 14 presents a Numberjack model of the Warehouse allocation problem. It consists of finding an optimal set of warehouses to maintain a number of businesses. Each warehouse has a fixed cost of remaining open and a maximum number of businesses it can supply. Each business has a transport cost for each of the warehouses. The

```
data = WareHouseData()

WareHouseOpen = VarArray(data.get("NumberOfWarehouses"))

ShopSupplied = Matrix(data.get("NumberOfShops"),
                      data.get("NumberOfWarehouses"))

# Cost of running warehouses
warehouseCost = Sum(WareHouseOpen, data.get("WareHouseCosts"))

# Cost of shops using warehouses
transpCost = Sum([ Sum(varRow, costRow) for (varRow, costRow) in
                   zip(ShopSupplied, data.get("SupplyCost"))])

obj = warehouseCost + transpCost

model = Model(
   # Objective function
   Minimise(obj),
   # Channel from store opening to store supply matrix
   [[var <= store for var in col] for (col, store) in
                   zip(ShopSupplied.col, WareHouseOpen)],
   # Make sure every shop if supplied by one store
   [Sum(row) == 1 for row in ShopSupplied.row],
   # Make sure that each store does not exceed its supply capacity
   [Sum(col) <= cap for (col, cap) in
                   zip(ShopSupplied.col, data.get("Capacity"))]
)
```

Fig. 14. A Numberjack model of the Warehouse Allocation problem.

Table 3. SCIP vs Mistral (Warehouse Allocation).

Instance	SCIP			Mistral		
	Objective	Nodes	Time	Objective	Nodes	Time
cap44.dat	1184690	1	0.84	1468957	10008044	>3600
cap63.dat	1087190	14	1.82	1388391	10683754	>3600
cap71.dat	957125	1	0.69	1297505	11029722	>3600
cap81.dat	811324	1	0.65	1409091	3497095	>3600
cap131.dat	954894	5	5.30	1457632	1281009	>3600

task is to find the optimal set of warehouses to open such that all the businesses are supplied and the total cost is minimal. This problem is easily solved by the Mixed Integer Solver SCIP which is part of Numberjack, conversely the Constraint Solver, Mistral struggles on the simplest of instances.

We report the results for five Warehouse allocation instances from the CSPLIB[10] in Table 3. In all cases, SCIP could find an optimal solution in a few nodes, whilst Mistral ran over a time limit of one hour, staying well over the optimal allocation. These examples showcase the advantage of being able to change solvers without changing the model. Being able to change the point of attack on a given problem so easily can provide insights into the structure of the problem and how best to solve it.

[10] http://www.csplib.org/ (prob 0034).

6 Conclusion

In this paper we present Numberjack, 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. We present the features of Numberjack through the use of several combinatorial problems from academia, from optical networking, jobshop scheduling, and warehouse allocation. We demonstrated the ease with which cloud-based applications could be created using Numberjack. Finally we demonstrated its performance on a variety of problems.

References

1. T. Achterberg, T. Berthold, T. Koch, and K. Wolter. Constraint integer programming: A new approach to integrate cp and mip. In *CPAIOR*, pages 6–20, 2008.
2. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting Systematic Search by Weighting Constraints. In Ramon López de Mntaras and Lorenza Saitta, editors, *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI-04)*, pages 482–486, Valencia, Spain, August 2004. IOS Press.
3. Alain Colmerauer. An introduction to prolog iii. *Commun. ACM*, 33(7):69–90, 1990.
4. Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abderrahmane Aggoun, and Alexander Herold. The chip system: Constraint handling in prolog. In Ewing L. Lusk and Ross A. Overbeek, editors, *CADE*, volume 310 of *Lecture Notes in Computer Science*, pages 774–775. Springer, 1988.
5. Jacob Feldman, Eugene C. Freuder, and James Little. Cp-inside: Embedding constraint-based decision engines in business applications. In Willem Jan van Hoeve and John N. Hooker, editors, *CPAIOR*, volume 5547 of *Lecture Notes in Computer Science*, pages 323–324. Springer, 2009.
6. D. Grimes, E. Hebrard, and A. Malapert. Closing the Open Shop: Contradicting Conventional Wisdom. In Ian Gent, editor, *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP-09)*, Lecture Notes in Computer Science, pages 400–408, Lisbon, Portugal, September 2009. Springer-Verlag.
7. Pascal Van Hentenryck, Laurent Michel, Laurent Perron, and Jean-Charles Régin. Constraint programming in opl. In Gopalan Nadathur, editor, *PPDP*, volume 1702 of *Lecture Notes in Computer Science*, pages 98–116. Springer, 1999.
8. J.N. Hooker. *Integrated Methods for Optimization (International Series in Operations Research & Management Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
9. 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 *Transparent Optical Networks, 2005, Proceedings of 2005 7th International Conference*, volume 1, pages 433–436 Vol. 1, July 2005.
10. N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. Minizinc: Towards a standard cp modelling language. In *CP*, pages 529–543, 2007.
11. Jean-Francois Puget. Constraint programming next challenge: Simplicity of use. In Mark Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 5–8. Springer, 2004.
12. N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear csp into sat. *Constraints*, 14(2):254–272, 2009.
13. Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005.