Constraining order Documentation

Release 0.1

Johannes Reinhardt

January 12, 2015

Contents

Constraining Order is a pure python library for solving certain classes of constraint satisfaction problems (CSP). In particular it contains an implementation of IntervalSets to represent fairly general sets of real numbers.

Constraining Order works (at least) with Python 2.7, 3.4 and PyPy and is licensed under the MIT license.

Constraining Order is relatively flexible in the sense that many CSPs can be quickly modeled and new constraints can be implemented very easily. However it is not tuned towards high performance.

For serious problems there are much more powerful solutions available:

- gecode (which looks amazing and superbly documented)
- or-tools
- choco
- or one of the many specialized constraint programming languages

For python there are several packages for constraint satisfaction problems:

- ortools
- gecode-python outdated, inactive
- logilab-constraints sparse documentation, inactive
- pyconstraints sparse documentation, inactive

For a nice overview over the theoretical foundations, see e.g. [Tsang96].

The code is hosted on GitHub:

https://github.com/jreinhardt/constraining-order

Contents:

CHAPTER 1

Quickstart

For a quick introduction we will write a Sudoku solver. Sudokus are popular puzzles that are often found in Newspapers. The task is to fill in numbers in a (already partly filled) 9x9 grid, such that no number is present twice in a row, column or one of the 9 3x3 blocks.

Sudokus are constraint satisfaction problems. The 81 fields of the grid are the variables, their domain is the set of numbers from 1 to 9, and the constraints are the number placement rules.

1.1 Variables

The first step is to model the space our problem lives in. For that we need the variables and their domains. We can make use of python to do this very compactly for all 81 variables

```
from constrainingorder.sets import DiscreteSet
from constrainingorder.variables import DiscreteVariable

numbers = range(1,10)
domain = DiscreteSet(numbers)
variables = {}
for i in numbers:
    for j in numbers:
        name = 'x%d%d' % (i,j)
        variables[name] = DiscreteVariable(name,domain=domain)
```

A DiscreteSet is a datastructure representing a set of discrete elements, very similar to pythons built-in set. But it can also represent the "set of everything", which is sometimes convenient. For details see .. todo:: Add DiscreteSet reference

A DiscreteVariable is a variable that can take on values from a DiscreteSet. Each variable has a name. The variables *x11* is the number in the first row and first column, the variable *x12* the one in the first row and second column and so on. We store them in a dictionary, so that we can easily to refer to them by name when building the constraints. This is often convenient, but not always necessary.

1.2 Constraints

The constraints model the requirements, that no number is allowed to occur twice in a row, column or block. Or equivalently, that all numbers in a row, column or block are different (as there are exactly nine different numbers). Luckily constraining order already comes with a constraint of this type, so we just have to use it:

```
from constrainingorder.constraints import AllDifferent

cons = []
#row constraints
for i in numbers:
    cons.append(AllDifferent([variables['x%d%d'%(i,j)] for j in numbers]))
#column constraints
for i in numbers:
    cons.append(AllDifferent([variables['x%d%d'%(j,i)] for j in numbers]))
#block constraints
for i in range(0,3):
    for j in range(0,3):
        #assemble list of parameternames for this block
        names = []
        for k in range(0,3):
            for l in range(0,3):
                names.append('x%d%d' % (3*i + k + 1,3*j + l + 1))
        #create constraint
        cons.append(AllDifferent([variables[n] for n in names]))
```

If we wanted to find all possible completely filled sudokus, we could now try to enumerate all solutions to this problem (see below), but this would take a very, very long while, as there are 6.67 10^21 different ones see [Felgenhauer].

In the sudokus found in newspapers some numbers are already given, in such a way that there is only one solution. We can add these filled in numbers by adding additional constraints that restrict certain variables to just a single value. Again this kind of constraint is already included in Constraining Order:

```
from constrainingorder.constraints import FixedValue

cons.append(FixedValue(variables['x11'],1))
cons.append(FixedValue(variables['x14'],8))
cons.append(FixedValue(variables['x21'],6))
cons.append(FixedValue(variables['x22'],3))
cons.append(FixedValue(variables['x25'],5))
cons.append(FixedValue(variables['x27'],9))
cons.append(FixedValue(variables['x32'],9))
cons.append(FixedValue(variables['x36'],3))
cons.append(FixedValue(variables['x37'],5))
cons.append(FixedValue(variables['x44'],2))
cons.append(FixedValue(variables['x47'],6))
cons.append(FixedValue(variables['x49'],3))
cons.append(FixedValue(variables['x51'],3))
cons.append(FixedValue(variables['x53'],2))
cons.append(FixedValue(variables['x57'],1))
cons.append(FixedValue(variables['x59'],7))
cons.append(FixedValue(variables['x61'],9))
cons.append(FixedValue(variables['x63'],8))
cons.append(FixedValue(variables['x66'],6))
cons.append(FixedValue(variables['x73'],6))
cons.append(FixedValue(variables['x74'],5))
cons.append(FixedValue(variables['x78'],7))
cons.append(FixedValue(variables['x83'],9))
cons.append(FixedValue(variables['x85'],6))
cons.append(FixedValue(variables['x88'],2))
cons.append(FixedValue(variables['x89'],5))
cons.append(FixedValue(variables['x96'],8))
cons.append(FixedValue(variables['x99'],9))
```

1.3 Space

With the variables and the constraints we can set up a Space. A Space collects all the variables and constraints, and keeps track of the possible values (the domain) for each variable. We print the domain for the first few variables.

```
from constrainingorder import Space

space = Space(variables.values(),cons)
for vname, domain in sorted(space.domains.items())[:15]:
    print vname, domain
```

This outputs

```
x11 {1,2,3,4,5,6,7,8,9}
x12 {1,2,3,4,5,6,7,8,9}
x13 {1,2,3,4,5,6,7,8,9}
x14 {1,2,3,4,5,6,7,8,9}
x15 {1,2,3,4,5,6,7,8,9}
x16 {1,2,3,4,5,6,7,8,9}
x17 {1,2,3,4,5,6,7,8,9}
x18 {1,2,3,4,5,6,7,8,9}
x19 {1,2,3,4,5,6,7,8,9}
x21 {1,2,3,4,5,6,7,8,9}
x22 {1,2,3,4,5,6,7,8,9}
x23 {1,2,3,4,5,6,7,8,9}
x24 {1,2,3,4,5,6,7,8,9}
x25 {1,2,3,4,5,6,7,8,9}
x26 {1,2,3,4,5,6,7,8,9}
```

A space can also tell us if a labelling (a dictionary with parameter names and values) is consistent with the constraints or satisfies them.

1.4 Solution

With the Space set up, we can now solve the CSP with backtracking, i.e. by filling in a number into a field and then checking if this is consistent with the constraints. If it is put a number into another field, if not, try another number, or if all numbers have been tried, go back to the previous field and try another number there.

This procedure can take a long time, as there are 9^81 possibilities that have to be tried. One possibility to speed this up is to reduce the problem space. For some fields possible numbers can be eliminated, as they are not consistent with the posed constraints. For example if the value of a field is fixed to 3, then its value can not be something else, and also the 3 can be removed from the domain of the fields in the same row, column and block.

In the constraint satisfaction literature this is called problem reduction. Constraining Order has an algorithm included for problem reduction called ac3, that does that.

```
from constrainingorder.solver import ac3

ac3(space)
for vname, domain in sorted(space.domains.items())[:15]:
    print vname, domain
```

Which now yields

```
x11 {1}
x12 {2,4,5,7}
x13 {4,5,7}
```

```
x14 {8}
x15 {2,4,7,9}
x16 {2,4,7,9}
x17 {2,3,4,7}
x18 {3,4,6}
x19 {2,4,6}
x21 {6}
x22 {3}
x23 {4,7}
x24 {1,4,7}
x25 {5}
x26 {1,2,4,7}
```

We can see that the domains of the variables have been reduces dramatically, which will speed up backtracking by a huge factor. Another thing that has a big impact on the performance is the order in which the variables are tried. In general one wants find conflicts as early as possible, as this eliminates whole branches of the search tree at once. For the case of sudoku a columns wise ordering (or row or blockwise) has proven to be effective.

Finally we can solve the sudoku by backtracking. The solve function is a generator which iterates over all found solutions. In this case we only want one, so break out of the loop after the first one is found.

```
from constrainingorder.solver import solve

#column wise ordering
ordering = []
for i in numbers:
    for j in numbers:
        ordering.append('x%d%d' % (i,j))

#find first solution and print it, then stop
for solution in solve(space,method='backtrack',ordering=ordering):
    for i in numbers:
        for j in numbers:
            print solution['x%d%d' % (i,j)],
        print
    break
```

The output of the solution should look like this

```
1 2 5 8 9 4 7 3 6
6 3 7 1 5 2 9 4 8
8 9 4 6 7 3 5 1 2
4 5 1 2 8 7 6 9 3
3 6 2 9 4 5 1 8 7
9 7 8 3 1 6 2 5 4
2 4 6 5 3 9 8 7 1
7 8 9 4 6 1 3 2 5
5 1 3 7 2 8 4 6 9
```

1.5 References

CHAPTER 2

Sets

In addition to constraint satisfaction, Constraining Order also contains an implementation of Intervals and IntervalSets over the real numbers and datastructures for sets in several dimensions.

2.1 DiscreteSet

The use of the `DiscreteSet` is very similar to the built in `frozenset`:

```
>>> from constrainingorder.sets import DiscreteSet
>>> a = DiscreteSet([1,2,'a','b'])
>>> b = DiscreteSet([1,'a','c',3])
```

DiscreteSets support the usual set operations and membership tests

```
>>> a.union(b)
DiscreteSet([1,2,3,'a','b','c'])
>>> a.intersection(b)
DiscreteSet([1,'a'])
>>> a.difference(b)
DiscreteSet([2,'b'])
>>> 1 in a
True
>>> "Foo" in b
False
```

The main difference is that a DiscreteSet can represent a set of "everything", which makes sense sometimes

```
>>> c = DiscreteSet.everything()
>>> c.union(a)
DiscreteSet.everything()
>>> c.intersection(a)
DiscreteSet([1,2,'a','b'])
```

One can also iterate over all members

```
>>> [m for m in a.iter_members()]
[1, 2, 'a', 'b']
```

2.2 Interval

To initialize a Interval one passes the bounds and indicates whether they are included in the interval, or alternatively one of the convenience class methods

```
>>> from constrainingorder.sets import Interval
>>> a = Interval((0,1),(True,True))
>>> b = Interval.open(1,3)
>>> c = Interval.leftopen(2,4)
```

Intervals only support the intersection operation, as for the others the result might not be a single connected interval.

```
>>> b.intersection(c)
Interval((2, 3),(False, False))
```

One can check membership in Intervals

```
>>> 0.3 in a
True
>>> 1.3 in a
False
```

Intervals can also represent the full real axis and a single point:

```
>>> d = Interval.everything()
>>> e = Interval.from_value(2.4)
```

2.3 IntervalSets

The main use of Intervals is in IntervalSets, which can represent fairly general sets of real numbers. They get initialized by a sequence of Intervals, or one of the convenience functions

```
>>> from constrainingorder.sets import Interval,IntervalSet
>>> a = IntervalSet([Interval.open(0,3), Interval.open(5,8)])
>>> b = IntervalSet([Interval.closed(2,3), Interval.closed(7,10)])
>>> c = IntervalSet.from_values([4, -1])
>>> d = IntervalSet.everything()
```

In contrast to Intervals, IntervalSets support all of the common set operations

```
>>> a.union(b)
IntervalSet([Interval((0, 3),(False, True)),Interval((5, 10),(False, True))])
>>> a.intersection(b)
IntervalSet([Interval((2, 3),(True, False)),Interval((7, 8),(True, False))])
>>> a.difference(b)
IntervalSet([Interval((0, 2),(False, False)),Interval((5, 7),(False, False))])
```

Membership tests work as expected

```
>>> 2 in a
True
>>> 4 in a
False
>>> -1 in c
True
```

Like DiscreteSets, one can iterate over the members if the IntervalSet only contains isolated points

```
>>> c.is_discrete()
True
>>> [m for m in c.iter_members()]
[-1, 4]
```

CHAPTER 3

Custom constraints

Constraining Order is designed to make it easy to add custom constraints. This tutorial will show this for the example of one of the most prominent constraint satisfaction problems, the 8 queens problem.

3.1 8 queens problem

The task is to place 8 queens on a chessboard in such a way that no queen can attack any other queen, i.e. no two queens occupy the same row, column or same diagonals.

One way to model this is by using 8 variables, one for a queen in each column. In this way, the requirement that there has to be one queen in every column is already built in, which reduces the number of constraints and the domain of the variables and improves performance.

As values for the variables we choose tuples with the actual coordinates on the board, this makes it easier to formulate the diagonal constraints. We name the variables with lowercase letter, the traditional naming schema of the columns of a chess board. Coordinates will be zero indexed, as this is more convenient in python.

```
from constrainingorder.sets import DiscreteSet
from constrainingorder.variables import DiscreteVariable

variables = {}
for i,col in enumerate('abcdefgh'):
    domain = DiscreteSet([(j,i) for j in range(8)])
    variables[col] = DiscreteVariable(col,domain=domain)
```

As we already built in the column constraint, it remains to express the row and diagonal constraints. We will take care of all of them by creating a new constraint type, a QueensConstraint:

```
from constrainingorder.constraints import Constraint
from itertools import product

class QueensConstraint(Constraint):
    """
    Constraint that ensures that a number of queens on a chessboard can
    not attack each other
    """
    def __init__(self,queens):
        """
        Create a new Queens constraint.

        :param variable: Variables representing the position of queens on a chess board
        :type variable: list of DiscreteVariables
```

```
        """
        Constraint.__init__(self,dict((var,var.domain) for var in queens))

    def _conflict(self,val1,val2):
        #check for row conflict
        if val1[0] == val2[0]:
            return True
        #check for rising diagonal conflict
        if val1[0] - val1[1] == val2[0] - val2[1]:
            return True
        #check for falling diagonal conflict
        if val1[0] + val1[1] == val2[0] + val2[1]:
            return True
    def satisfied(self,lab):
        for v1,v2 in product(self.vnames,repeat=2):
            if v1 == v2:
                continue
            if v1 not in lab or v2 not in lab:
                return False
            if self._conflict(lab[v1],lab[v2]):
                return False
        return True
    def consistent(self,lab):
        for v1,v2 in product(self.vnames,repeat=2):
            if v1 not in lab or v2 not in lab or v1 == v2:
                continue
            if self._conflict(lab[v1],lab[v2]):
                return False
        return True
```

A constraint needs to derive from the `Constraint` class and implement the two methods *satisfied* and *consistent*.

In the constructor we pass a dictionary of variables and the values for them which are consistent with this constraint. In this case, there is no field on the board excluded a priori, so we use the full domain of the variable.

As both of the methods we have to implement check for conflicts between the queens, it makes sense to write a small utility method that does this check to avoid code duplication. It compares the first component of the tuples to check for a row conflict. To check whether the two queens are on the same diagonal, we compare the sum and difference of the row and column components. It might not be obvious, but it is easy to check that fields with the same sum or difference of rows and columns are on the same diagonal. Not that we don't check for column conflicts, as they are taken care of by the setup of our variables.

The *satisfied* method checks that the labelling (a dictionary with variable names and values) assigns values to all variables affected by this constraint, and that there are no conflicts. The parameter names of the affected variables are accessible in the attribute `vnames`, that the Constraint class sets up for us.

The *consistent* method is a bit weaker, as it just checks for conflicts, but doesn't care about missing values. It allows the solution and reduction algorithms to detect inconsistencies even if not all queens are placed yet.

And thats it. We can now use this constraint just like the in-built ones:

```
from constrainingorder import Space
from constrainingorder.solver import solve

constraint = QueensConstraint(variables.values())
space = Space(variables.values(),[constraint])


for solution in solve(space,method='backtrack'):
    for i in range(8):
```

```
        for j in range(8):
            if (i,j) in solution.values():
                print 'X',
            else:
                print '.',
        print
    break
```

In contrast to the sudoku solver discussed in the *Quickstart*, the 8 queens problem space can not be reduced, as no fields can be eliminated a priori, for every field there exist solutions where a queen occupies this field.

We also don't specify a variable ordering, as in this case the total number of variables is rather low, and solution is quick in any case.

```
X . . . . . . .
. . . . . X . .
. . . . . . . X
. . X . . . . .
. . . . . . X .
. . . X . . . .
. X . . . . . .
. . . . X . . .
```

3.2 Custom Binary relations

A riddle from this weeks newspaper:

> Professor Knooster is visiting Shroombia. The people of Shroombia is divided into two groups, the shrimpfs that always lie and the wex that always tell the truth. For his research the professor asked 10 Shroombians about their groups. The answers:
>
> - Zeroo: Onesy is a shrimpf
> - Onesy: Threesy is a shrimpf
> - Twoo: Foursy is a shrimpf
> - Threesy: Sixee is a shrimpf
> - Foursy: Seveen is a shrimpf
> - Fivsy: Ninee is a shrimpf
> - Sixee: Twoo is a shrimpf
> - Seveen: Eightsy is a shrimpf
> - Eightsy: Fivsy is a shrimpf
>
> The professor sighed: "I will never find out who is in which group if you continue like this." Then the last Shroombian answered
>
> - Ninee: Zeroo and Sixee belong to different groups

This riddle can be modelled as a CSP, and it gives the opportunity to discuss a special kind of constraint, namely binary relations.

First set up the variables

```
from constrainingorder.variables import DiscreteVariable
from constrainingorder.sets import DiscreteSet

domain = DiscreteSet(['Shrimpf','Wex'])
variables = []
for i in range(10):
    variables.append(DiscreteVariable(str(i),domain=domain))
```

So every variable represents one Shroombian, who can be either a shrimpf or a wex.

Almost all hints are of the same structure: one Shroombian accuses another Shroombian of being a shrimpf. The hint is fulfilled either if the accusing shroombian is a Shrimpf (who is always lying) and the accused shroombian is not actually a Shrimpf, or if the accusor is a Wex (who is always telling the truth) and the accused is a in fact a Shrimpf.

We can represent this in form of a custom constraint. As each hint affects two shroombians, such constraints are binary relations. The implementation of binary relations is much simpler than for general constraints.

```
from constrainingorder.constraints import BinaryRelation

class Accusation(BinaryRelation):
    def relation(self,val1,val2):
        return (val1 == 'Shrimpf' and val2 == 'Wex') or\
               (val1 == 'Wex' and val2 == 'Shrimpf')
```

For classes derived from BinaryRelations it suffices to implement a single method that returns True if the two values fulfill the relation and False otherwise. Specific constraints are obtained by instantiating this class with two variables.

For DiscreteVariables with small domains one can represent binary relations also by listing all tuples of values that fulfill the relation. An equivalent implementation would be derived from DiscreteBinaryRelation.

```
from constrainingorder.constraints import DiscreteBinaryRelation

class Accusation(DiscreteBinaryRelation):
    def __init__(self,var1,var2):
        DiscreteBinaryRelation.__init__(self,var1,var2,[
            ('Shrimpf','Wex'), ('Wex','Shrimpf')
        ])
```

In addition we need to implement a new constraint for the last hint. As it affects three shroombians, this is not a binary relation.

```
from constrainingorder.constraints import Constraint
class AllegedNonEqual(Constraint):
    def __init__(self,var1,var2,var3):
        Constraint.__init__(self,{
            var1 : var1.domain,
            var2 : var2.domain,
            var3 : var3.domain}
        )
        self.v1 = var1.name
        self.v2 = var2.name
        self.v3 = var3.name

    def satisfied(self,lab):
        if not (self.v1 in lab and self.v2 in lab and self.v3 in lab):
            return False
        elif lab[self.v1] == 'Shrimpf':
            return lab[self.v2] == lab[self.v3]
        elif lab[self.v1] == 'Wex':
            return lab[self.v2] != lab[self.v3]
```

```
    def consistent(self,lab):
        if not (self.v1 in lab and self.v2 in lab and self.v3 in lab):
            return True
        elif lab[self.v1] == 'Shrimpf':
            return lab[self.v2] == lab[self.v3]
        elif lab[self.v1] == 'Wex':
            return lab[self.v2] != lab[self.v3]
```

Now we can specify the constraints

```
cons = []
cons.append(Accusation(variables[0],variables[1]))
cons.append(Accusation(variables[1],variables[3]))
cons.append(Accusation(variables[2],variables[4]))
cons.append(Accusation(variables[3],variables[6]))
cons.append(Accusation(variables[4],variables[7]))
cons.append(Accusation(variables[5],variables[9]))
cons.append(Accusation(variables[6],variables[2]))
cons.append(Accusation(variables[7],variables[8]))
cons.append(Accusation(variables[8],variables[5]))

cons.append(AllegedNonEqual(variables[9],variables[0],variables[6]))
```

And solve the problem

```
from constrainingorder import Space
from constrainingorder.solver import solve

space = Space(variables,cons)

for solution in solve(space,method='backtrack'):
    for name, group in sorted(solution.items()):
        print name, group
```

```
0 Shrimpf
1 Wex
2 Shrimpf
3 Shrimpf
4 Wex
5 Shrimpf
6 Wex
7 Shrimpf
8 Wex
9 Wex
```

API Reference:

CHAPTER 4

Sets

Constraining Order contains DataStructures to represent sets of discrete elements and real numbers.

A DiscreteSet is a wrapper around pythons builtin `frozenset`. The main difference is that a DiscreteSet can represent a set of all possible elements.

In addition, there are data structures to represent sets of real numbers, in form of connected `Intervals` and collections of such intervals, called `IntervalSet`.

4.1 DiscreteSet

class `constrainingorder.sets.`**`DiscreteSet`**(*elements*)

A set data structure for hashable elements

This is a wrapper around pythons set type, which additionally provides the possibility to express the set of everything (which only makes sense sometimes).

`__contains__`(*element*)

Check membership of the element.

Parameters **element** – Element to check membership of

Return type bool

`__init__`(*elements*)

Create a new DiscreteSet

Parameters **elements** (*sequence*) – The elements of the newly created set

`difference`(*other*)

Return a new DiscreteSet with the difference of the two sets, i.e. all elements that are in self but not in other.

Parameters **other** (*DiscreteSet*) – Set to subtract

Return type DiscreteSet

Raises **ValueError** if self is a set of everything

classmethod **`everything`**()

Create a new set of everything.

One can not iterate over the elements of this set, but many operations are actually well defined and useful.

intersection(*other*)
Return a new DiscreteSet with the intersection of the two sets, i.e. all elements that are in both self and other.

Parameters **other** (*DiscreteSet*) – Set to intersect with

Return type DiscreteSet

is_discrete()
Check whether the set is discrete, i.e. if `iter_members()` can be used.

Return type bool

is_empty()
Check whether the set is empty

Return type bool

iter_members()
Iterate over all elements of the set.

Raises ValueError if self is a set of everything

union(*other*)
Return a new DiscreteSet with the union of the two sets, i.e. all elements that are in self or in other.

Parameters **other** (*DiscreteSet*) – Set to unite with

Return type DiscreteSet

4.2 Interval

class constrainingorder.sets.**Interval**(*bounds*, *included*)
An interval on the real axis.

__contains__(*x*)
Check membership of the element.

Parameters **x** (*float*) – Element to check membership of

Return type bool

__init__(*bounds*, *included*)
Create a new Interval with bounds. If the right bound is larger than the left bound, the interval is assumed to be empty.

Parameters

- **bounds** (*sequence*) – left and right bounds
- **included** (*sequence*) – bools indicating whether the bounds are included in the interval.

classmethod closed(*a*, *b*)
Create a new closed Interval.

Parameters

- **a** (*float*) – Left bound
- **b** (*float*) – Right bound

classmethod everything()
Create a new Interval representing the full real axis

classmethod **from_value**(*value*)
Create a new Interval representing a single real number.

Parameters **value** (*float*) – The member of the Interval

get_point()
Return the number contained in this interval.

Return type float

Raises ValueError if Interval contains more than exactly one number.

intersection(*other*)
Return a new Interval with the intersection of the two intervals, i.e. all elements that are in both self and other.

Parameters **other** (*Interval*) – Interval to intersect with

Return type Interval

is_discrete()
Check whether this interval contains exactly one number

Return type bool

is_disjoint(*other*)
Check whether two Intervals are disjoint.

Parameters **other** (*Interval*) – The Interval to check disjointedness with.

is_empty()
Check whether this interval is empty.

Return type bool

classmethod **leftopen**(*a*, *b*)
Create a new halfopen Interval (left bound is excluded, right bound included).

Parameters

- **a** (*float*) – Left bound
- **b** (*float*) – Right bound

classmethod **open**(*a*, *b*)
Create a new open Interval.

Parameters

- **a** (*float*) – Left bound
- **b** (*float*) – Right bound

classmethod **rightopen**(*a*, *b*)
Create a new halfopen Interval (right bound is excluded, left bound included).

Parameters

- **a** (*float*) – Left bound
- **b** (*float*) – Right bound

4.3 IntervalSet

class constrainingorder.sets.**IntervalSet**(*ints*)

A set of intervals to represent quite general sets in R

__contains__(*x*)

Check membership of the element.

Parameters **element** – Element to check membership of

Return type bool

__init__(*ints*)

Create a new IntervalSet.

Parameters **ints** (*sequence*) – Intervals for this IntervalSet

difference(*other*)

Return a new IntervalSet with the difference of the two sets, i.e. all elements that are in self but not in other.

Parameters **other** (*IntervalSet*) – Set to subtract

Return type IntervalSet

classmethod everything()

Create a new IntervalSet representing the full real axis.

classmethod from_values(*values*)

Create a new IntervalSet representing a set of isolated real numbers.

Parameters **values** (*sequence*) – The values for this IntervalSet

intersection(*other*)

Return a new IntervalSet with the intersection of the two sets, i.e. all elements that are both in self and other.

Parameters **other** (*IntervalSet*) – Set to intersect with

Return type IntervalSet

is_discrete()

Check whether this IntervalSet contains only isolated numbers.

Return type bool

is_empty()

Check whether this IntervalSet is empty.

Return type bool

iter_members()

Iterate over all elements of the set.

Raises ValueError if self is a set of everything

union(*other*)

Return a new IntervalSet with the union of the two sets, i.e. all elements that are in self or other.

Parameters **other** (*IntervalSet*) – Set to intersect with

Return type IntervalSet

CHAPTER 5

Constraint Satisfaction

5.1 Variables

Variables are derived from a common baseclass

class constrainingorder.variables.**Variable**(*name*, ***kwargs*)
Abstract baseclass for variables.

Variables describe the variables of a CSP. The instances are immutable and only make sense in connection with a Space.

description = None
description of the variable

discrete = None
whether the variable is discrete or continuous

domain = None
domain of the variable

name = None
name of the variable

Constrainingorder at the moment contains two types of Variables, DiscreteVariables and RealVariables

class constrainingorder.variables.**DiscreteVariable**(*name*, ***kwargs*)
Discrete variable with values from a DiscreteSet of elements.

__init__(*name*, ***kwargs*)
Create a new DiscreteVariable

Parameters

- **name** (*str*) – The name of the variable
- **description** (*str*) – An optional description of the variable
- **domain** (*DiscreteSet*) – An optional domain for this variable, defaults to everything.

class constrainingorder.variables.**RealVariable**(*name*, ***kwargs*)
Continuous real variable with values from the real numbers.

__init__(*name*, ***kwargs*)
Create a new RealVariable

Parameters

- **name** (*str*) – The name of the variable

- **description** (*str*) – An optional description of the variable
- **domain** (*IntervalSet*) – An optional domain for this variable, defaults to everything.

5.2 Constraints

Constraints are derived from a common baseclass

class constrainingorder.constraints.**Constraint**(*domains*)

consistent(*lab*)
check whether the labeling is consistent with this constraint

Parameters **lab** (*dict*) – A dictionary with parameter names and values

Return type bool

domains = None
Domains imposed by node consistency for this constraint

satisfied(*lab*)
check whether the labeling satisfies this constraint

Parameters **lab** (*dict*) – A dictionary with parameter names and values

Return type bool

vnames = None
Names of the variables affected by this constraint

Constrainingorder ships with a selection of constraints, but it is easy to add custom ones

class constrainingorder.constraints.**FixedValue**(*variable*, *value*)
Constraint that fixes a variable to a value

__init__(*variable*, *value*)
Create a new FixedValue constraint. It enforces that a variable takes on a particular, fixed value.

Parameters

- **variable** (*Variable*) – Variable whose value is fixed
- **value** – Value to which it is fixed

Raises ValueError if the value is not in the domain of the variable

class constrainingorder.constraints.**AllDifferent**(*variables*)
Constraint enforcing different values between a number of variables

__init__(*variables*)
Create a new AllDifferent constraint. It enforces that a set of variable takexs on different values.

Parameters **variables** (*sequence*) – Variables for this Constraint

class constrainingorder.constraints.**Domain**(*variable*, *domain*)
Constraint that ensures that value of a variable falls into a given domain

__init__(*variable*, *domain*)
Create a new Domain constraint. It enforces that a variable takes on values from a specified set.

Parameters

- **variable** (*DiscreteVariable or RealVariable*) – Variable whose value is restricted

- **domain** (*DiscreteSet or IntervalSet*) – Set of values to which variable is restricted

Binary relations are an important class of constraints. In Constraining Order they are derived from a common base-class. New binary relations only need to implement the relation function. These relations can be used on Variables with values that offer the corresponding relations in the python data model.

class constrainingorder.constraints.**BinaryRelation**(*var1*, *var2*)
Abstract Base class for constraint the describe a binary relation between two variables.

__init__(*var1*, *var2*)
Create a new binary relation constraint between these two variables

Parameters

- **var1** (*DiscreteVariable or RealVariable*) – The first variable
- **var2** (*DiscreteVariable or RealVariable*) – The second variable

relation(*val1*, *val2*)
evaluate the relation between two values

Parameters

- **val1** – The value of the first variable
- **val2** – The value of the second variable

Return type bool

Constraining Order ships with the standard relations.

class constrainingorder.constraints.**Equal**(*var1*, *var2*)
Equality relation

class constrainingorder.constraints.**NonEqual**(*var1*, *var2*)
Inequality relation

class constrainingorder.constraints.**Less**(*var1*, *var2*)
Smaller-than relation

class constrainingorder.constraints.**LessEqual**(*var1*, *var2*)
Smaller or equal relation

class constrainingorder.constraints.**Greater**(*var1*, *var2*)
Larger-than relation

class constrainingorder.constraints.**GreaterEqual**(*var1*, *var2*)
Larger or equal relation

For DiscreteVariables, another way to represent relations is by the set of tuples that are fulfilling this relation. This is represented by the DiscreteBinaryRelation constraint

class constrainingorder.constraints.**DiscreteBinaryRelation**(*var1*, *var2*, *tuples*)
General binary relation between discrete variables represented by the tuples that are in this relation

__init__(*var1*, *var2*, *tuples*)
Create a new DiscreteBinaryRelation constraint. It restricts the values of the two variables to a set of possible combinations.

Parameters

- **var1** (*DiscreteVariable or RealVariable*) – The first variable
- **var2** (*DiscreteVariable or RealVariable*) – The second variable
- **tuples** (*sequence of tuples with values*) – The allowed value combinations

5.3 Space

class constrainingorder.**Space**(*variables*, *constraints*)
A space is a description of the computation space for a specific CSP.

__init__(*variables*, *constraints*)
Create a new Space for a CSP

Parameters

- **variables** (*sequence of Variables*) – The variables of the CSP
- **constraints** (*sequence of Constraints*) – The constraints of the CSP

consistent(*lab*)
Check whether the labeling is consistent with all constraints

constraints = None
list of constraints

domains = None
dictionary of variable names to DiscreteSet/IntervalSet with admissible values

is_discrete()
Return whether this space is discrete

satisfied(*lab*)
Check whether the labeling satisfies all constraints

variables = None
dictionary of variable names to variable instances

5.4 Solvers

To obtain one or all solutions to a CSP, one needs to use a solver. Solvers operate on a space. For good performance it might be good to reduce the problem space first.

constrainingorder.solver.**ac3**(*space*)
AC-3 algorithm. This reduces the domains of the variables by propagating constraints to ensure arc consistency.

Parameters **space** (*Space*) – The space to reduce

constrainingorder.solver.**solve**(*space*, *method=u'backtrack'*, *ordering=None*)
Generator for all solutions.

Parameters

- **method** (*str*) – the solution method to employ
- **ordering** (*sequence of parameter names*) – an optional parameter ordering

Methods:

"backtrack" simple chronological backtracking

"ac-lookahead" full lookahead

CHAPTER 6

Indices and tables

- *genindex*
- *modindex*
- *search*

CHAPTER 7

References

[Felgenhauer] Bertram Felgenhauer and Frazer Jarvis. Enumerating possible sudoku grids. Technical report, 2005

[Tsang96] Tsang, E. Foundations of Constraint Satisfaction Academic Press, 1996

Symbols

A

B

C

D

E

F

G