Python-CPSolver reference manual

September 2024

Contents

1 Installing **2**

2 Solving a CSP **2**

2.1 How to use . . . 2

2.2 Variables and Expressions . . . 3

2.3 Constraints . . . 3

2.3.1 Specific constraints . . . 3

2.3.2 Generic constraint . . . 4

2.3.3 Special factions . . . 4

2.4 Searching . . . 4

2.4.1 Seeking satisfaction . . . 4

2.4.2 Optimizing . . . 5

2.4.3 Getting solutions . . . 5

2.5 Output . . . 5

1 Installing

Prerequisites: To use this software, you only need Python 3. It can be easily installed directly from the Python Package Index (PyPI) and works out of the box.

```
pip install PythonCPSolver
```

Additionally, since this software is designed for evaluating new algorithms, the source code is available on GitHub:

https://github.com/GonzaloHernandez/python-cpsolver

2 Solving a CSP

$CSP = (X, (D_i)_{i \in X}, C)$

cpsolver: $CSP \rightarrow \{(x_i, d_i) \mid x_i \in X, d_i \in D_i, \forall C_j \in C : C_j \text{ } \textit{is satisfied} \text{ }\}$

2.1 How to use

Python-CPSolver consists of four core modules: engine, brancher, propagators, and variables. In most scenarios, importing the entire package is adequate for utilizing Python-CPSolver. A basic program typically includes five main sections:

```
- <package>
- <decision_variables> [See section 2.2]
- <constraints> [See section 2.3]
- <seraching> [See section 2.4]
- <output> [See section 2.5]
```

```
# package
from PythonCPSolver import *

# decision variables
x = IntVar(1,5)
y = IntVar(3,8)
z = IntVar(1,10)

# constraints
constraint1 = AllDifferent([x,y,z])
constraint2 = Constraint(x+y == z)

# searching
engine = Engine( [x,y,z], [constraint1,constraint2] )
solutions = engine.search()

# output
for s in solutions :  print( toInts(s) )
```

This example shows the first solution found: $(x_1 = 1, x_2 = 3, x_3 = 4)$.

2.2 Variables and Expressions

Python-CPSolver uses a generic integer variable type with bounds $-2147483647 \leqslant x \leqslant 2147483647$, where the parameters primarily set the limits.

```
IntVar(min:int, max:int, name:int ) :  IntVar
```

```
x1 = IntVar(1,10,"var")
x2 = IntVar(-5,4)
x3 = IntVar(3)
x4 = IntVar()
```

This example instanciates x_1 labeld as *var*, and x_2, x_3, x_4 without labels, where $(1 \leqslant x_1 \leqslant 10)$, $(-5 \leqslant x_2 \leqslant 4)$, $(3 \leqslant x_3 \leqslant 2147483647)$ and $(-2147483647 \leqslant x_4 \leqslant 2147483647)$.

```
IntVarArray(n:int, min:int, max:int, prefix:str ) :  IntVar[]
```

```
V = IntVarArray(5,1,10,"var")
W = IntVarArray(3)
```

This example creates two sets of variables, $V = \{v_i \ \mid 0 \leqslant i < 5 \ and \ 1 \leqslant v_i \leqslant 10\}$ labeled as *{var0, var1, .. var4}* and $W = \{w_i \ \mid 0 \leqslant i < 3 \ and \ -2147483647 \leqslant w_i \leqslant 2147483647\}$ without labels.

```
Expression( expr:Expression ) :  Expression [Automatic construction]
```

```
ex1 = x1*(x2-6)
ex2 = V[2] >= 4
```

This example shows two mathematical expression useful to filter the searching and to define optimization functions. The operators suported in current version are: $+, -, *, ==, != , <, >, <=, >=, \&$ *and* $|$.

2.3 Constraints

Python-CPSolver implements several constraints, with its own propagators enforcing these restrictions during the search process.

2.3.1 Specific constraints

```
AllDifferent( vars:list ) :  Constraint
```

```
c1 = AllDifferent( W )
c2 = AllDifferent( [ x1,x3,V[2] ] )
```

This example implement two constraints. First, Constraint C_1 asserts that $\bigwedge_{w,v \in W, w \neq v} w \neq v$. Second, constraint C_2 asserts that *alldifferent*(x_1, x_3, v_2).

```
Linear( vars:list, total:IntVar ) :  Constraint
```

```
c3 = Linear( V, 5 )
c4 = Linear( [x2,x3,x4], W[0] )
```

This example implement two constraints. First, Constraint C_3 asserts that $\Sigma_{v \in V}(v) = 5$. Second, constraint C_4 asserts that $(x_2 + x_3 + v_4) = w_0$. This version only support equalities.

```
LinearArgs( args:list, vars:list, total:IntVar ) :  Constraint
```

```
c5 = LinearArgs( [5,4,2], [x1,x2,x3], 12 )
c6 = LinearArgs( [2,2,2], W, x4 )
```

This example implement two constraints. First, Constraint C_5 asserts that $(5x_1 + 4x_2 + 2x_3) = 12$. Second, constraint C_6 asserts that $(2w_0 + 2w_1 + 2w_2) = x_4$.

2.3.2 Generic constraint

```
Constraint( expr:Expression ) :  Constraint
```

```
c7 = Constraint( x1+x4 > ex1 )
c8 = Constraint( x4 == x3+x2 )
```

This example implement two constraints. First, given that $ex1 := x_1 * (x_2 - 6)$ then constraint C_7 asserts that $(x_1 + x_4) > (x_1 * (x_2 - 6))$. Second, constraint C_8 asserts that $x_4 = (x_3 + x_2)$.

2.3.3 Special factions

```
count( vars:list, eqcond:Expression ) :  Expression
```

```
c9 = Constraint( count(V,3) > x2 )
```

This example implement a constraint C_9 to assert that $(|\{v \in V | v = 3\}| > x_2)$.

```
sum( vars:list ) :  Expression
```

```
c10 = Constraint( x1*3 == sum(W) )
c11 = Constraint( sum([x1,x2,x3]) > sum(V) )
```

This example implement two constraints. First, The constraint C_{10} asserts that $((x_1 * 3) = \Sigma_{w \in W}(w))$. Second, constraint C_{11} asserts that $(\Sigma_{1 \leqslant i \leqslant 3}(x_i) > \Sigma_{v \in V}(v))$.

2.4 Searching

Python-CPSolver includes an Engine designed to perform searches using the backtracking methodology, combined with branch and bound techniques to optimize the search process.

2.4.1 Seeking satisfaction

```
Engine( vars:list, cons:list ) :  Engine
```

```
e1 = Engine( [x1,x2,x3,x4], [c7,c8] )
e2 = Engine( W, [c1] )
e3 = Engine( V )
```

This example implements three search engines. First, it creates e_1 to explore the set of decision variables $\{x_1, x_2, x_3, x_4\}$ while enforcing the condition $((x_1+x_4) > (x_1 \cdot (x_2-6))) \wedge (x_4 = (x_3+x_2))$. Second, it create e_2 to explore the set W ensuring that $\mathit{alldifferent}(w_0, w_1, w_2)$ holds. Third, the engine e_3 will enumerate all combinations of domains for $v \in V$ without any constraints.

2.4.2 Optimizing

```
Engine( vars:list, cons:list, func:[OPTYPE,Expression] ) :  Engine
```

```
e4 = Engine( [x1,x2,x3,x4], [c7,c8], minimize(x3+x4) )
```

This example enhances the engine e_4 by building upon e_3 to find the solution that minimizes the calculation of $x_3 + x_4$. The available functions ar:

```
minimize( exp:Expression ) :  [MINIMIZE,exp]
maximize( exp:Expression ) :  [MAXIMIZE,exp]
```

2.4.3 Getting solutions

```
Engine.search( top:int ) :  list[]
```

```
s1 = e4.search()
s2 = e1.search(3)
s3 = e2.search(ALL)
```

In this example, the solutions from the search are stored in a list of fixed variables. First, it stores the only solution found by e_4, represented as $s_1 = [[x'_1, x'_2, x'_3, x'_4]]$, Second, It stores the top three solutions found by e_1, wich are represented as $s_2 = [[w'_0, w'_1, w'_2], [w''_0, w''_1, w''_2], [w'''_0, w'''_1, w'''_2]]$. Lastly, s_3 contains all the solutions found by e_2. When using optimization functions, the paramenter is ignored.

2.5 Output

Since the solver stores solutions as IntVar types, there are two functions available to convert them into primitive types.

```
toInts( vars:list ) :  list
toStrs( vars:list ) :  list
```

```
print( toInts(s1[0]))
```

This example prints the only solution stored in s_1, converted into a list of integer values.

```
for s in s2 :  print( toInts(s) )
```

This example prints each solution stored in s_2, converting them into lists of integer values.

```
for s in s3 :  print( toStrs(s,IntVar.PRINT_VALUE) )
```

This example prints the whole set of solution stored in s_3, converting them into lists of strings (only values). There are other modifiers:

```
PRINT_NAME
```

To print only the name of the variable

```
PRINT_MIX
```

To print both name + value