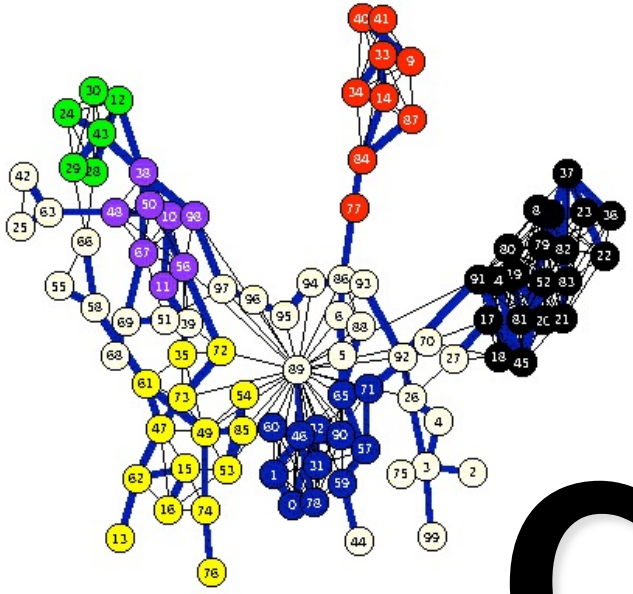


CSP in Python

Overview

- **Python-constraint** is a good package for solving CSP problems in Python
- Installing it
- Using it
- Examples in
  - Magic Squares
  - Map coloring
  - Sudoku puzzles
  - HW?: Battleships

Installation

- On your own computer
  - pip install python-constraint
  - sudo pip install python-constraint
- Install locally on gl
  - pip3 install –user python-constraints
- Install locally on UMBC Jupyter hub server by executing this once in a notebook
  - !pip install –user python-constraints
- Clone source from github
  - https://github.com/python-constraint

Simple Example

```
>>> from constraint import *
>>> p = Problem()
>>> p.addVariable("a", [1,2,3])
>>> p.addVariable("b", [4,5,6])
>>> p.getSolutions()
[{'a': 3, 'b': 6}, {'a': 3, 'b': 5}, {'a': 3, 'b': 4},
 {'a': 2, 'b': 6}, {'a': 2, 'b': 5}, {'a': 2, 'b': 4},
 {'a': 1, 'b': 6}, {'a': 1, 'b': 5}, {'a': 1, 'b': 4}]

>>> p.addConstraint(lambda x,y: 2*x==y, ('a','b'))
>>> p.getSolutions()
[{'a': 3, 'b': 6}, {'a': 2, 'b': 4}]
```

variable name

domain

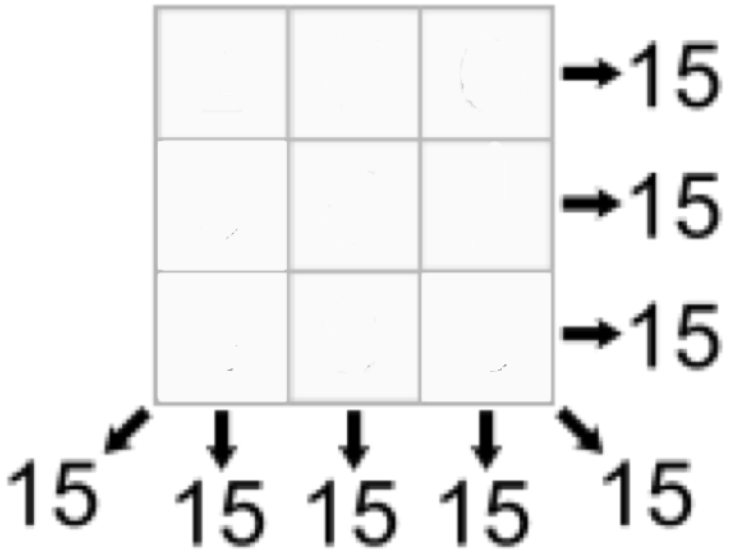
9 solutions
(instantiations)

two variables

constraint function

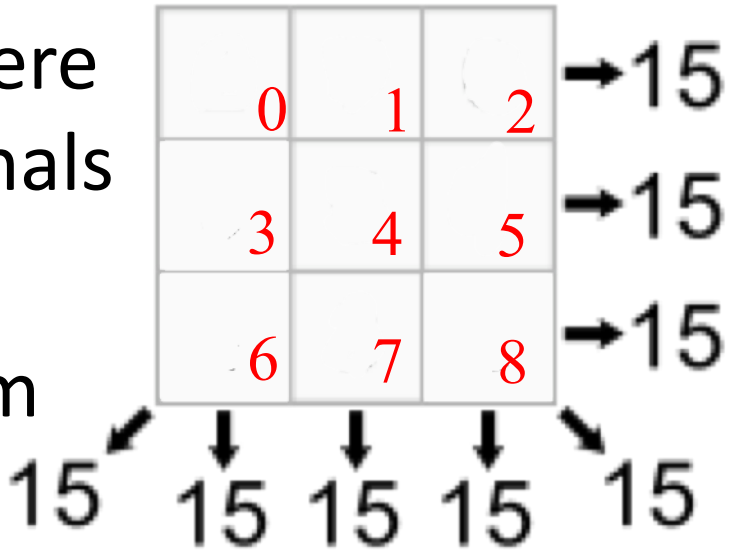
Magic Square

- An NxN array of integers where all rows, columns and diagonals sum to the same number
- Given N (e.g., 3) and magic sum (e.g., 15), find cell values
- What are the
  - Variables & their domains
  - Constraints



Magic Square

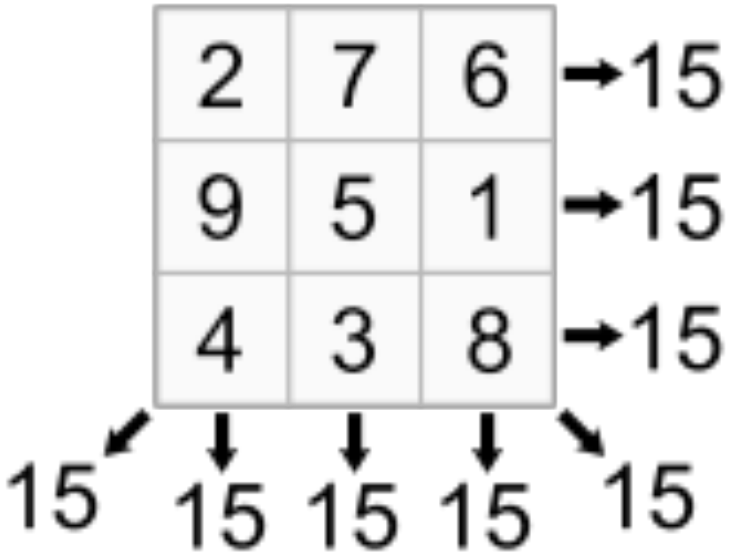
- An NxN array of integers where all rows, columns and diagonals sum to the same number
- Given N (e.g., 3) & magic sum (e.g., 15), find cell values
- What are the
  - **Variables [0..8]** & their **domains [1..9]**
  - Constraints



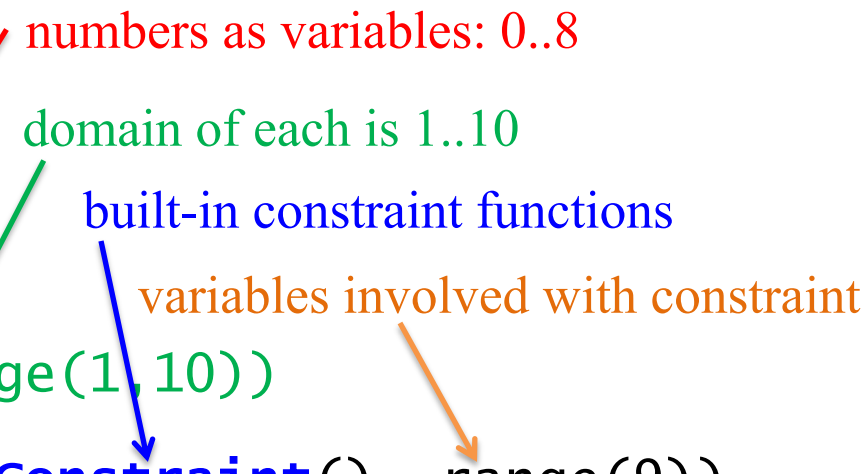
All variables have different values
v0+v1+v2 ==15, v0+v3+v6 == 15, …

Magic Square

- An NxN array of integers where all rows, columns and diagonals sum to the same number
- Given N (e.g., 3) and the magic sum (e.g., 15) find the cell values
- What are the
  - Variables & their domains
  - Constraints



3x3 Magic Square




```
from constraint import *

p = Problem()

p.addVariables(range(9), range(1,10))

p.addConstraint(AllDifferentConstraint(), range(9))

p.addConstraint(ExactSumConstraint(15), [0,4,8])
p.addConstraint(ExactSumConstraint(15), [2,4,6])
for row in range(3):
    p.addConstraint(ExactSumConstraint(15),
                    [row*3+i for i in range(3)])
for col in range(3):
    p.addConstraint(ExactSumConstraint(15),
                    [col+3*i for i in range(3)])
```

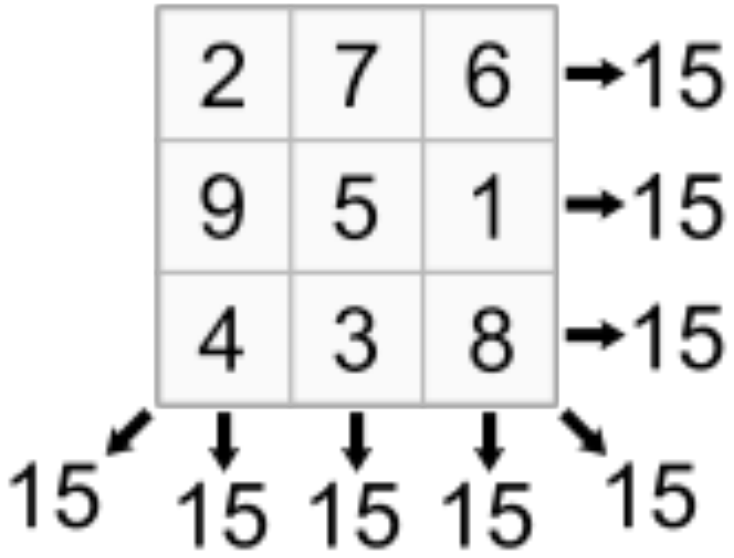
3x3 Magic Square

```
sols = p.getSolutions()
print sols

for s in sols:
    print
    for row in range(3):
        for col in range(3):
            print s[row*3+col],
        print
```

3x3 Magic Square

```
> python ms3.py
[{0:6,1:7,2:2,…8:4}, {0:6,1:…}, …]
```

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

6 1 8
7 5 3
2 9 4
… six more solutions …
```



Constraints

- FunctionConstraint(f, v)
- Arguments:
  - F: a function of N (N>0) arguments
  - V: a list of N variables
- Function can be defined & referenced by name or defined locally via lambda expressions
  - `p.addConstraint(lambda x,y:x==2*y,[11,22])`
  - `def dblfn(x,y): return x == 2*y`

    `P.addConstraint(dblfn, [11,22])`

Constraints

- Constraints on a set of variables:
  - AllDifferentConstraint()
  - AllEqualConstraint()
  - MaxSumConstraint()
  - ExactSumConstraint()
  - MinSumConstraint()
- Examples:

```
p.addConstraint(ExactSumConstraint(100),[11,…19])
p.addConstraint(AllDifferentConstraint(),[11,…19])
```

Constraints

- Constraints on a set of possible values
  - InSetConstraint()
  - NotInSetConstraint()
  - SomeInSetConstraint()
  - SomeNotInSetConstraint()

Map Coloring

- For map coloring, each country is a variable and the domains are the set of available colors
- Constraints: countries sharing a boarder can't have the same color
- A simple python string can encode
  - Strings to use as variables for the countries
  - Counties sharing a border
- "**SA**:WA NT Q NSW V; **NT**:WA Q; **NSW**: Q V; **T**:"
  - "sharing a border" is symmetric, so only mention it once

Map Coloring

```
def color(map, colors=['red','green','blue']):
    (vars, adjoins) = parse_map(map)
    p = Problem()
    p.addVariables(vars, colors)
    for (v1, v2) in adjoins:
        p.addConstraint(lambda x,y: x!=y, [v1, v2])
    solution = p.getSolution()
    if solution:
        for v in vars:
            print "%s:%s " % (v, solution[v]),
        print
    else:
        print 'No solution found :-('

austrailia = "SA:WA NT Q NSW V; NT:WA Q; NSW: Q V; T:"
```

Map Coloring

```
australia = 'SA:WA NT Q NSW V; NT:WA Q; NSW: Q V; T:’

def parse_map(neighbors):
    adjoins = []
    regions = set()
    specs = [spec.split(':') for spec in neighbors.split(';')]
    for (A, Aneighbors) in specs:
        A = A.strip();
        regions.add(A)
        for B in Aneighbors.split():
            regions.add(B)
            adjoins.append([A,B])
    return (list(regions), adjoins)
```

Sudoku

```
def sudoku(initValue):
    p = Problem()
    # Define a variable for each cell: 11,12,13...21,22,23...98,99
    for i in range(1, 10) :
        p.addVariables(range(i*10+1, i*10+10), range(1, 10))
    # Each row has different values
    for i in range(1, 10) :
        p.addConstraint(AllDifferentConstraint(), range(i*10+1, i*10+10))
    # Each colum has different values
    for i in range(1, 10) :
        p.addConstraint(AllDifferentConstraint(), range(10+i, 100+i, 10))
    # Each 3x3 box has different values
    p.addConstraint(AllDifferentConstraint(), [11,12,13,21,22,23,31,32,33])
    p.addConstraint(AllDifferentConstraint(), [41,42,43,51,52,53,61,62,63])
    p.addConstraint(AllDifferentConstraint(), [71,72,73,81,82,83,91,92,93])

    p.addConstraint(AllDifferentConstraint(), [14,15,16,24,25,26,34,35,36])
    p.addConstraint(AllDifferentConstraint(), [44,45,46,54,55,56,64,65,66])
    p.addConstraint(AllDifferentConstraint(), [74,75,76,84,85,86,94,95,96])

    p.addConstraint(AllDifferentConstraint(), [17,18,19,27,28,29,37,38,39])
    p.addConstraint(AllDifferentConstraint(), [47,48,49,57,58,59,67,68,69])
    p.addConstraint(AllDifferentConstraint(), [77,78,79,87,88,89,97,98,99])

    # add unary constraints for cells with initial non-zero values
    for i in range(1, 10) :
        for j in range(1, 10):
            value = initValue[i-1][j-1]
            if value: p.addConstraint(lambda var, val=value: var == val, (i*10+j,))

    return p.getSolution()
```

Sudoku Input

```
easy = [[0,9,0,7,0,0,8,6,0],
        [0,3,1,0,0,5,0,2,0],
        [8,0,6,0,0,0,0,0,0],
        [0,0,7,0,5,0,0,0,6],
        [0,0,0,3,0,7,0,0,0],
        [5,0,0,0,1,0,7,0,0],
        [0,0,0,0,0,0,1,0,9],
        [0,2,0,6,0,0,0,5,0],
        [0,5,4,0,0,8,0,7,0]]
```