Résolution des jeux de réflexion: prise en main du solveur

Pascale Le Gall - Jean-Philippe Poli

Centrale-Supélec

2024-2025

python constraint

Tout au long de cette séance, nous allons utiliser en guise de solveur, python constraint.

Python constraint est un module qui offre un solveur pédagogique pour la résolution de problèmes à satisfaction de contraintes.

Dans notre cas, le solveur va nous permettre de définir un CSP en python et savoir si celui-ci a une ou plusieurs solutions (s'il est satisfiable).

Installation

Utilisez pip pour installer la librairie [1] :

```
pip install python-constraint
```

1. Plus de détails : `https://pypi.org/project/python-constraint/`.

Création du problème

Importez le module constraint : **from** constraint **import** $*$

Dans Python Constraint, les notions de problème et de solveur sont mélangées. Tout commence par la création d'un problème :

```
problem = Problem()
```

Ajout de variables au problème

Avant toute chose, il faut déclarer les variables. Nous avons vu en cours que chaque variable était définie sur un domaine donné et énumérable. La méthode addVariable prend en paramètre le nom de la variable, sous forme d'une chaîne de caractères, et le domaine énumérable, sous la forme d'une séquence en python.
Une séquence en python peut être un tuple, une liste, ou un énumérateur comme celui renvoyé par la fonction **range**() que vous connaissez déjà.
Une chaîne de caractères est aussi une séquence de caractères.

problem.addVariable(nom de la variable, domaine de la variable)

Si vous souhaitez une variable x telle que $x \in [0; 9]$, il faudra écrire :
problem.addVariable("x", **range**(10))

Ajout de variables au problème

Il est possible d'ajouter plusieurs variables ayant le même domaine avec la méthode :
problem.addVariables(liste des noms de variables , liste des valeurs).

Si vous souhaitez trois variables x, y, z sur $[0; 9]$, il faudra écrire :
problem.addVariables (["x" , "y" , "z"], **range**(10))

Ajout de contraintes au problème

Python constraint dispose de différents types de contraintes.

Contraintes principales

- AllDifferentConstraint : les valeurs des variables concernées doivent être toutes différentes
- AllEqualConstraint : les valeurs des variables concernées doivent être toutes égales
- ExactSumConstraint : la somme des valeurs des variables concernées doit être égale à la valeur indiquée
- MaxSumConstraint : la somme des valeurs des variables concernées doit être au plus égale à la valeur indiquée
- MinSumConstraint : la somme des valeurs des variables concernées doit être au moins égale à la valeur indiquée

Ajout de contraintes au problème

Ajout d'une contrainte au problème

L'ajout d'une contrainte se fait par la fonction addConstraint().

Exemple

problem.addConstraint(AllDifferentConstraint (), ["x", "y", "z"]) indique que les variables x, y, z préalablement ajoutées au problème doivent avoir des valeurs différentes.

Notez bien :

- les parenthèses après le type de contraintes
- que le second argument est une liste de variables, même si elle ne doit contenir qu'un élément.

Ajout de contraintes au problème

Python constraint offre également la possibilité d'utiliser une fonction comme contrainte. Cette fonction a autant de paramètres qu'elle ne concerne de variables, et retourne forcément un booléen (True ou False).

Contrainte fonction

Soit une fonction egalite définie comme suit :

```
def egalite(a,b):
        return a==b
```

Alors elle peut être utilisée comme contrainte comme suit :
problem.addConstraint(egalite , ["x", "y"]).

Notez bien la différence entre les noms des paramètres de la fonction (a, b) et les noms des variables du problème ("x", "y"). Python constraint utilise l'ordre dans lequel vous donnez les variables du problème pour faire l'appariement entre "x" et a, et "y" et b.

Ajout de contraintes au problème

En Python, il y a un moyen très rapide de déclarer des petites fonctions. On appelle ça des fonctions lambda.
En mathématiques, on déclarerait : $egalite : a, b \mapsto a == b$ ce qui donne en python : **lambda** a,b:a==b.

Contrainte fonction lambda

La contrainte précédente peut donc s'écrire plus simplement :
problem.addConstraint(**lambda** a,b:a==b, ["x", "y"]).

Cela n'a pas d'impact sur la performance de la résolution, mais réduit la taille du code. N'utilisez cette syntaxe que si vous êtes à l'aise avec Python.

Résolution du problème

Une fois le problème problem créé et les contraintes ajoutées, nous allons pouvoir en demander la résolution.

Pour obtenir la première solution du solveur, utilisez :
`problem.getSolution();`.
Pour obtenir toutes les solutions du solveur, utilisez :
`problem.getSolutions();`.

Manipulation des listes

Pour pouvoir faire certains exercices, il faudra utiliser des listes de noms de variables.

Pour créer une liste de variables nommées V1, ..., V15, vous pouvez écrire :

```
liste = ['V%i'%(i) for i in range(1,16)]
```

ce qui est équivalent à :

```
liste = []
for i in range(1,16):
        liste.append('V%i'%(i))
```

mais bien plus concis.

Il n'est malheureusement pas possible d'utiliser des listes variables dans les contraintes de comparaison. Par exemple, vous ne pouvez pas comparer deux listes de variables.

Coloration de la carte d'Australie

Enoncé

Colorier les provinces d'Australie avec 3 couleurs (rouge, vert, bleu) de telle sorte que 2 provinces adjacentes n'aient pas la même couleur.

Coloration de la carte d'Australie

Principe

- Déclarer une variable pour chaque province,
- Chaque variable est définie sur le domaine "*bleu*","*vert*","*rouge*",
- On ajoute une contrainte $\neq$ entre deux variables correspondant à des provinces adjacentes.

Coloration de la carte d'Australie

```
from constraint import *

problem = Problem()
```

Coloration de la carte d'Australie

```
from constraint import *

problem = Problem()

problem.addVariables(["WA", "NT", "SA", "Q", "NSW", "V", "T"], ["Rouge", "Vert", "Bleu"])
```

Coloration de la carte d'Australie

```
from constraint import *

problem = Problem()

problem.addVariables(["WA", "NT", "SA", "Q", "NSW", "V", "T"], ["Rouge", "Vert", "Bleu"])

problem.addConstraint(AllDifferentConstraint(),["WA","NT"])
problem.addConstraint(AllDifferentConstraint(),["WA","SA"])
problem.addConstraint(AllDifferentConstraint(),["NT","SA"])
problem.addConstraint(AllDifferentConstraint(),["NT","Q"])
problem.addConstraint(AllDifferentConstraint(),["SA","Q"])
problem.addConstraint(AllDifferentConstraint(),["SA","NSW"])
problem.addConstraint(AllDifferentConstraint(),["SA","V"])
problem.addConstraint(AllDifferentConstraint(),["Q","NSW"])
problem.addConstraint(AllDifferentConstraint(),["NSW","V"])
```

Coloration de la carte d'Australie

```
print(problem.getSolutions())
```

COCA+COLA=OASIS

Enoncé

Nous nous intéressons au cryptarithme suivant :

$$\begin{array}{r} \ C\ O\ C\ A \\ +\ \ C\ O\ L\ A \\ \hline O\ A\ S\ I\ S \end{array}$$

Il s'agit d'associer chaque lettre à un chiffre différent entre 0 et 9 de telle sorte que l'opération puisse s'effectuer.

COCA+COLA=OASIS

Formulation 1

- Déclarer une variable pour chaque lettre,
- Chaque variable est définie sur le domaine $[0; 9]$, sauf les variables qui sont en début de ligne : O et C.
- On utilise la contrainte AllDifferent
- On utilise une contrainte spécifiant que :
  $2 \times C \times 1000 + 2 \times O \times 100 + C \times 10 + L \times 10 + 2 \times A =$
  $O \times 10000 + A \times 1000 + S \times 100 + I \times 10 + S$

COCA+COLA=OASIS

```
from constraint import *

problem = Problem()
problem.addVariables(["C", "O", "L", "A", "S", "I"], range(10))

def c1(c, o, l, a, s, i):
    return 1000 * c + 100 * o + 10 * c + a + 1000 * c + 100 * o + 10 * l + a
                        == 10000 *o + 1000 * a + 100 * s + 10 * i + s

problem.addConstraint(c1, ("C", "O", "L", "A", "S", "I") )
problem.addConstraint(AllDifferentConstraint())
print(problem.getSolutions())
```

COCA+COLA=OASIS

Formulation 2

- Déclarer une variable pour chaque lettre,
- Chaque variable est définie sur le domaine $[0; 9]$, sauf les variables qui sont en début de ligne : O et C.
- On utilise la contrainte AllDifferent
- On pose les contraintes verticalement, par exemple : $2 \times A = S$
- Attention aux retenues : il faut 3 variables supplémentaires valant soit 0 soit 1

$$\begin{array}{cccccc} & & r_3 & r_2 & r_1 & \\ & & C & O & C & A \\ + & & C & O & L & A \\ \hline & O & A & S & I & S \end{array}$$

COCA+COLA=OASIS

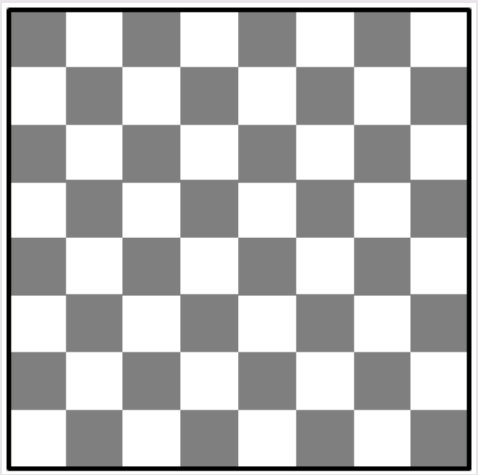
```
problem = Problem()
problem.addVariables(["C", "O", "L", "A", "S", "I"], range(10))
problem.addVariables(["R1", "R2", "R3"], range(2))

def AplusAegalS(a, s, r1):
    return s == 2*a - r1*10

def CplusLegalI(c, l, i, r1, r2):
    return i == r1+c+l-r2*10

def OplusOegalS(o, s, r2, r3):
    return s == 2*o + r2 - r3*10

def CplusCegalA(c, a, o, r3):
    return a == 2*c+r3 - 10*o

problem.addConstraint(AplusAegalS, ("A", "S", "R1") )
problem.addConstraint(CplusLegalI, ("C", "L", "I", "R1", "R2") )
problem.addConstraint(OplusOegalS, ("O", "S", "R2", "R3") )
problem.addConstraint(CplusCegalA, ("C", "A", "O", "R3") )
problem.addConstraint(AllDifferentConstraint(), ("C", "O", "L", "A", "S", "I"))
print(problem.getSolutions())
```

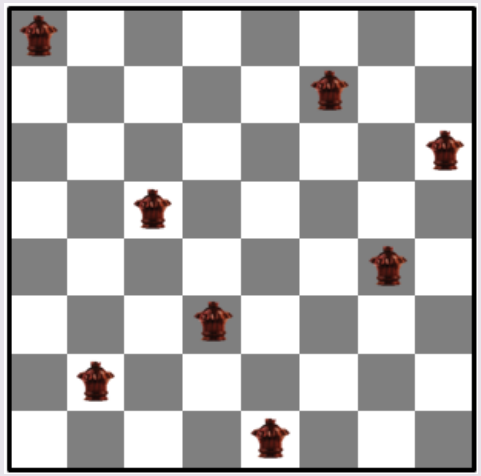
Les n reines

Enoncé

Soit un plateau d'échec de dimension $n \times n$. Il s'agit de placer n reines telles qu'elles ne puissent se manger les unes les autres (vis-à-vis des règles des échecs).
Autrement dit, il ne faut pas deux reines sur la même ligne, la même colonne ou la même diagonale.

Exemple de plateau pour les 8 reines et une solution





Les n reines

Formulation 1

- Déclarer une variable pour chaque case du plateau,
- Chaque variable indique la présence d'une reine sur cette case, donc soit 0 (absence) soit 1 (présence)
- On vérifie la présence d'une seule reine sur chaque ligne, colonne et diagonale
- On utilise ExactSumConstraint()

Les n reines : formulation 1

```
p = Problem()

#Each variable represents a cell
Q = [["Q_%i_%i"%(i+1, j+1) for j in range(n)] for i in range(n)]
for row in Q:
                    p.addVariables(row, range(2))
```

Les n reines : formulation 1

```
p = Problem()

#Each variable represents a cell
Q = [["Q_%i_%i"%(i+1, j+1) for j in range(n)] for i in range(n)]
for row in Q:
                p.addVariables(row, range(2))


#pas 2 reines sur la même ligne
for row in Q:
        p.addConstraint(ExactSumConstraint(1), row)
```

Les n reines : formulation 1

```
p = Problem()

#Each variable represents a cell
Q = [["Q_%i_%i"%(i+1, j+1) for j in range(n)] for i in range(n)]
for row in Q:
                p.addVariables(row, range(2))


#pas 2 reines sur la même ligne
for row in Q:
        p.addConstraint(ExactSumConstraint(1), row)


#pas 2 reines sur la même colonne
for col in range(n):
        p.addConstraint(ExactSumConstraint(1), [ Q[row][col] for row in range(n) ])
```

Les n reines : formulation 1

```
#diagonales montantes
for line in range(2,2*n-1):
                firstcol = max(0,line-n)
                count = min(line, (n-firstcol))
                diag = [Q[min(n,line)-j-1][firstcol+j] for j in range(count)]
                p.addConstraint(MaxSumConstraint(1), diag)
```

Les n reines : formulation 1

```
#diagonales montantes
for line in range(2,2*n-1):
                firstcol = max(0,line-n)
                count = min(line, (n-firstcol))
                diag = [Q[min(n,line)-j-1][firstcol+j] for j in range(count)]
                p.addConstraint(MaxSumConstraint(1), diag)


#diagonales descendantes
for col in range(2,2*n-1):
                firstline = max(n-col,0)
                count = min(n-firstline, 2*n-col)
                diag = [Q[firstline+j][max(0, col-n)+j] for j in range(count)]
                p.addConstraint(MaxSumConstraint(1), diag)
```

Les n reines

Dans la seconde formulation, on aura moins de variables, moins de contraintes, et des contraintes plus facilement exprimables.

Formulation 2

- Déclarer une variable par reine $Q_1, \ldots, Q_n$, Q_i représentant la reine sur la ième colonne,
- Chaque variable indique la ligne sur laquelle est placée la reine, donc une valeur entre 1 et n
- Il n'est plus utile de vérifier la présence d'une seule reine sur une même colonne
- Pour vérifier que toutes les reines sont sur des lignes différentes : AllDifferent
- Il reste que les diagonales à vérifier

Les n reines : formulation 2

```
p = Problem()
#Each variable represents the queen at column i
Q = ["Q_%i"%(i+1) for i in range(n)]
p.addVariables(Q, range(1,n+1))
```

Les n reines : formulation 2

```
#pas 2 reines sur la même ligne
p.addConstraint(AllDifferentConstraint())
```

Les n reines : formulation 2

```
#pas 2 reines sur la même ligne
p.addConstraint(AllDifferentConstraint())


#diagonales
def diagonals(i,j):
        """Qi et Qj ne sont pas sur la même diagonale"""
        def inner(Qi,Qj):
                        return fabs(Qi-Qj)!=fabs(i-j)

        return inner

for i in range(n):
        for j in range(i+1,n):
                        p.addConstraint(diagonals(i,j), (Q[i],Q[j]))
```