Ecole Polytechnique, Cycle ingénieur
MAP572

Résolution de k-SAT par Monte-Carlo

```
# css style
from IPython.core.display import HTML
def css_styling():
    styles = open("./style/custom2.css").read()
    return HTML(styles)
css_styling()
```

```
# load the libraries
import matplotlib.pyplot as plt # 2D plotting library
import numpy as np              # package for scientific computing
import random
%matplotlib inline
```

Table des matières

Pour un problème sur un espace fini, les méthodes *Monte-Carlo Markov Chain* (MCMC) consistent à parcourir l'ensemble des solutions possibles de façon aléatoire mais astucieuse. On cherche ainsi à déterminer la solution optimale, ou proche de l'optimal.

L'objectif de ce TP est d'illustrer la puissance de la stratégie MCMC sur un problème particulier : le problème k-SAT en informatique théorique. C'est aussi un prétexte pour utiliser les matrices de transitions.

Une référence pour ce TP est :

[1] MITZENMACHER, Michael et UPFAL, Eli. Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis. Cambridge university press, 2017. (pages 156-159)

Le problème k-SAT

Le problème SAT en informatique théorique (aussi appelé problème de satisfaisabilité booléenne) est le problème de décision qui, étant donné une formule de logique booléenne, détermine s'il existe une assignation des variables qui rend la formule vraie.

Nous allons définir tous les termes et restreindre le contexte.

Ici on va spécifiquement s'intéresser au problème k-SAT. Soit k fixé (dans ce TP on va prendre uniquement $k \in \{2, 3\}$) et étant données n variables booléennes $x_1, x_2, \dots, x_n \in \{\text{Vrai}, \text{Faux}\}$, on considère les formules booléennes de la forme ($\vee$ signifie 'ou' et $\wedge$ signifie 'et')

$$(z_{1,1} \vee z_{1,2} \vee \cdots \vee z_{1,k}) \wedge \cdots \wedge (z_{M,1} \vee z_{M,2} \vee \cdots \vee z_{M,k})$$

où

- M est un entier quelconque
- pour chaque $1 \leq m \leq M$ et chaque $i \leq k$ on a
$$z_{m,i} \in \left\{x_1, x_2, \dots, x_n, \overline{x_1}, \overline{x_2}, \dots, \overline{x_n}\right\}.$$

(La notation $\overline{x}$ désigne la négation de x.) Chaque terme $(z_{m,1} \vee z_{m,2} \vee \cdots \vee z_{m,k})$ est appelée une clause.

Par exemple pour 2-SAT avec $n = 5$ variables, une formule à $M = 3$ clauses est donnée par

$$F = (x_1 \vee \overline{x_5}) \wedge (x_2 \vee x_3) \wedge (\overline{x_1} \vee x_2)$$

Une *affectation* est une fonction $x_1, x_2, \dots, x_n \in \{\text{Vrai}, \text{Faux}\}$. S'il existe une affectation qui rende F vraie on dit que F est satisfiable.

Dans le cas de l'exemple ci-dessus, une *affectation* des variables qui rende F vraie est :

$$x_1 = \text{Faux},\ x_2 = \text{Vrai},\ x_3 = \text{Faux},\ x_4 = \text{Faux},\ x_5 = \text{Faux}.$$

Spécifiquement, le problème k-SAT est de trouver un algorithme qui, étant donnée une formule F, trouve une affectation des variables qui rende F vraie (ou qui retourne `impossible` si une telle affectation n'existe pas). Le problème 2-SAT est polynomial alors que 3-SAT est NP-complet (voir par exemple : S.Perifel. Complexité algorithmique. Ellipses (2014))

> **Do it yourself.**
> Ecrire des fonctions `ClauseVraie(Clause,Affectation,nb_variables)` et `FonctionVraie(Formule,Affectation,nb_variables)` qui prennent en entrée des clauses ou formules et une affectation, et calcule si la clause/formule est vraie ou

pas.

On pourra représenter les formules sous la forme de liste :

```
Formule = [Clause1,Clause2,...,ClauseM]
```

où chaque `Clause` est de la forme

```
Clause = [z_1,...,z_k]
```

et chaque z_i est un booléen. Dans les clauses on peut numéroter les booléens de 0 à $n-1$ pour $x_1, \ldots, x_n$ et de n à $2n-1$ pour leurs négations. Pour l'exemple plus haut :

```
F=[[0,9],[1,2],[5,1]]
```

car (par exemple) $\overline{x_5}$ est codé par 9. L'affectation donnée en exemple est alors

```
[0,1,0,0,0]
```

```
####################################
# ---- tester une clause
####################################

def ClauseVraie(Clause,Affectation,nb_variables):
    #
    #
    return XXX

# Test
n=2
Clause1=[0,1]  # x_1 ou x_2
Clause2=[2,1]  # non(x_1) ou x_2
Clause3=[2,3]  # non(x_1) ou non(x_2)

Affectation1=[1,0] # x_1 vraie et x_2 fausse

print(ClauseVraie(Clause1,Affectation1,n)) # doit renvoyer True
print(ClauseVraie(Clause2,Affectation1,n)) # doit renvoyer False
print(ClauseVraie(Clause3,Affectation1,n)) # doit renvoyer True
```

```
####################################
# ---- tester une formule
####################################

def FormuleVraie(Formule,Affectation,nb_variables):
    #
    #
    return XXX

# test
Formule1=[Clause1,Clause2]
Formule2=[Clause1,Clause3]
Formule3=[Clause2,Clause3]
print([FormuleVraie(f,Affectation1,2) for f in [Formule1,Formule2,Formule3]]) # Doit re

# test
```

```
Formule=[[0,9],[1,2],[5,1]]
Affectation=[0,1,0,0,0]
print(FormuleVraie(Formule,Affectation,5)) # Doit renvoyer True
```

2-SAT : l'algorithme WalkSat

Le problème 2-SAT est polynomial, donc "simple" à résoudre. Nous allons voir qu'un algorithme probabiliste extrêmement naïf en vient (presque) à bout.

Algorithme WalkSat
entrées : Formule F à M clauses sur n variables.
paramètre : T entier
sortie :

- Si c'est possible, renvoyer une affectation qui rende F vraie
- Sinon, renvoyer `impossible` .

1. Initialisation : On tire une affectation `Affectation` $= (x_1, \dots, x_n)$ uniforme au hasard dans $\{\text{Vrai}, \text{Faux}\}^n$
2. Faire $2Tn^2$ fois :
   - 2a) Chercher la première clause non satisfaite
   - 2b) Dans cette clause, choisir une variable x_i uniformément au hasard et la changer de valeur : $x_i \leftarrow \text{non}(x_i)$.
   - 2c) Si `Affectation` rend la formule vraie, renvoyer `Affectation`
3. Renvoyer `impossible`

(La raison pour laquelle on écrit le nombre de boucles sous la forme $2Tn^2$ apparaîtra à la partie suivante.)

> **Remark.**
> **Attention** L'algorithme WalkSat n'est pas tout à fait correct. Lorsque WalkSat renvoie `impossible` , cela ne signifie pas forcément que la formule n'est pas satisfiable mais peut-être simplement que l'algorithme n'a pas cherché assez longtemps.

> **Do it yourself.**
> Ecrire une fonction `UneEtapeWalkSat(n_var,Formule,Affectation)` qui effectue une fois les opérations 2a)-2b)-2c) dans la description de l'Algorithme WalkSat.

```
def UneEtapeWalkSat(n_var,Formule,Affectation):
    #
    #
    return XXX
```

WalkSat : choix du paramètre T

Rappelons les propriétés suivantes de l'algorithme WalkSat appliqué à une formule F à n variables :

- Si F est fausse, l'algorithme renvoie `impossible`
- Si F est satisfiable
  - Avec une proba que l'on va noter $p(n, T, F, \mathbf{x})$ où $\mathbf{x} = (x_1, \dots, x_n)$ est l'affectation initiale, l'algorithme se trompe et renvoie `impossible`
  - Avec une proba $1 - p(n, T, F, \mathbf{x})$ il renvoie une affectation correcte.

On note

$$p_\star(n, T) = \max_{F \text{ satisfiable}, \mathbf{x}} p(n, T, F, \mathbf{x})$$

où $\mathbf{x} = (x_1, \dots, x_n)$ est l'affectation initiale.

Tout l'enjeu est d'avoir une bonne majoration de $p_\star(n, T)$ en fonction de T.

Fixons une formule F satisfiable, et fixons également une affectation $(y_1, \dots, y_n)$ correcte pour F. Pour $t \geq 0$ on note $(x_1^t, \dots, x_n^t)$ l'affectation de WalkSat à l'instant t.

Soit

$$D_t = \text{card}\{i \text{ tels que } x_i^t \neq y_i\}.$$

Le processus $(D_t)_t$ est un processus aléatoire à valeurs dans $\{0, 1, \dots, n\}$, lorsqu'il touche 0 alors on a trouvé une affectation pour F. On introduit les temps aléatoires :

$$\begin{aligned}\tau_D &= \min\{t, D_t = 0\},\\ \tau_E &= \min\{t, (x_1^t, \dots, x_n^t) \text{ est une affectation correcte.}\}\end{aligned}$$

Do it yourself.
(Théorie)

1. Démontrer que pour tout t,

$$\mathbb{P}(\tau_E \geq t) \leq \mathbb{P}(\tau_D \geq t).$$

2. Soit $(S_t)_t$ le processus défini de la façon suivante :

   - $S_0 = n$
   - Si $S_t = n$, alors $S_{t+1} = n - 1$
   - Si $S_t = 0$ alors $S_{t+1} = 0$
   - Sinon S_t vaut $S_t - 1$ ou $S_t + 1$ avec probabilité $1/2$, indépendamment du passé.

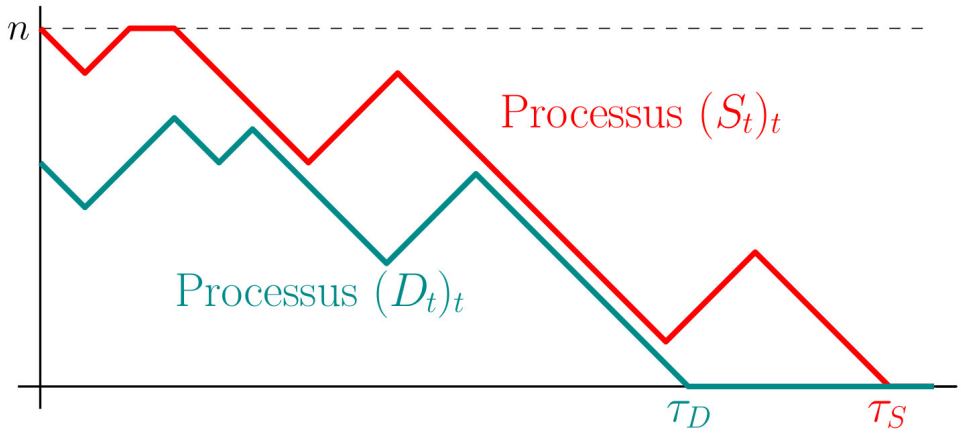
Ainsi (S_t) est la marche aléatoire symétrique sur l'intervalle $[0, n]$, réfléchie en n et absorbée en 0. On note

$$\tau_S = \min\{t, S_t = 0\}.$$

Démontrer que dans un certain sens τ_D a tendance à arriver plus tôt que τ_S. Formellement, pour tout t,

$$\mathbb{P}(\tau_D \geq t) \leq \mathbb{P}(\tau_S \geq t).$$

(Il n'est pas facile de rédiger soigneusement cette question, essayez plutôt de vous convaincre que c'est vrai!)



Answers.

1.
2.

Do it yourself. **(Théorie)**
Dans la section suivante vous allez vérifier numériquement que pour tout $n \geq 10$ alors

$$\mathbb{P}(\tau_S \leq 2n^2) \geq 0.89.$$

Grâce aux questions précédentes cela implique que

$$\mathbb{P}(\tau_E \leq 2n^2) \geq 0.89. \qquad (\star)$$

En utilisant $(\star)$, déterminer T pour que pour toute formule satisfiable F alors

$$\mathbb{P}(\text{WalkSat trouve une affectation pour } F) \geq 1 - \varepsilon.$$

Application numérique. Trouver T pour que

$$\mathbb{P}(\text{WalkSat trouve une affectation pour } F) \geq 99,99\%$$

Answers.

Do it yourself.
(Théorie) Mais au fait, pourquoi la stratégie WalkSat ne marche pas pour 3-SAT?

Answers.

Calcul de $\mathbb{P}(\tau_S \leq \lambda n^2)$ (matrices de transition)

Rappelons que (S_t) est la marche aléatoire symétrique partant de n, réfléchie en n et absorbée en 0. On note $\tau_S = \min\{t, S_t = 0\}$.

Pour déterminer T nous avons eu besoin d'estimer numériquement la probabilité $\mathbb{P}(\tau_S \leq 2n^2)$. Nous allons pour cela utiliser une matrice de transition. Pour $t \geq 0$ et $0 \leq i, j \leq n$ on note

$$p_{i,j}^{(t)} = \mathbb{P}(\text{sachant que } S_0 = i, \ldots)$$

Do it yourself.

1. Soit $0 \leq i \leq n$, $1 \leq j \leq n-1$ et $t \geq 0$. Justifier que
$$p_{i,j}^{(t)} = \frac{1}{2} p_{i,j-1}^{(t-1)} + \frac{1}{2} p_{i,j+1}^{(t-1)}.$$
2. Réfléchir rapidement au cas $j = 0$ ou $j = n$ dans l'équation suivante et en déduire qu'il existe une matrice Q_n de taille $(n+1) \times (n+1)$telle que pour tous t, i, j,
$$p_{i,j}^{(t)} = (Q_n^t)_{i,j}.$$
Cette matrice Q_n est appellée matrice de transition du processus (S_t).
3. Ecrire $\mathbb{P}(\tau_S \leq t)$ en fonction de Q_n. En déduire un code python qui calcule $\mathbb{P}(\tau_S \leq 2n^2)$ de façon exacte et vérifier sur un graphique que cette probabilité semble converger lorsque $n \to +\infty$.

Answers.

1.
2.

```
# Question 3
```

Application : Illustration de la transition de phase

Le problème 2-SAT aléatoire (lorsque les formules sont tirées aléatoirement et uniformément) présente un phénomène de *transition de phase*. Pour être plus formel nous introduisons quelques notations.

Il y a $\binom{2n}{2}^M$ formules différentes avec M clauses et n variables (on considère que l'ordre des clauses compte, mais pas l'ordre des variables dans une clause). Notons

$$p(n, M)$$

la probabilité qu'une formule aléatoire uniforme parmi les $\binom{2n}{2}^M$ formules différentes soit vraie. On a bien sûr $p(n, M)$ qui est décroissante en M (plus il y a de clauses plus c'est difficile d'être vraie).

Pour $c > 0$ on a, lorsque $n \to +\infty$:

$$p(n, c \times n) \to \begin{cases} 1 & \text{si } c < 1 \\ 0 & \text{si } c > 1 \end{cases}$$

Référence : GENT, Ian P. et WALSH, Toby. The SAT phase transition. In : *ECAI*. 1994. p. 105-109.

Do it yourself.

1. Ecrire une fonction `TirerClause(k,nb_variables)` qui prend en entrées deux entiers k, n et renvoie une clause de k-SAT à n variables, uniformément au hasard.
2. Ecrire une fonction `TirerFormule(k,nb_variables,M)` qui renvoie une formule de M clauses de k-SAT à n variables, uniformément au hasard.

(On considère qu'il y a $\binom{2n}{2}^M$ formules différentes avec M clauses et n variables : l'ordre des clauses compte, mais pas l'ordre des variables dans une clause.)

Do it yourself.

1. Ecrire un solveur WalkSat qui avec le paramètre T choisi à l'exercice précédent.
2. En utilisant des simulations et le solveur WalkSat, illustrer la transition de phase en $c = 1$.

```
# Question 1
```

```
# Question 2
```