TP2: Recherche exhaustive et retour sur trace

Ce sujet est constitué de quatre exercices, sur SAT, le Sudoku, les cryptarithmes et le problème des huit reines. Pour le premier exercice, un fichier à compléter est proposé, notamment pour la gestion des importations des fichiers de test. L'objectif de la séance est de faire trois exercices : les deux premiers et au moins un des deux derniers.

Exercice 1. *Problème* SAT

Objectifs de l'exercice. Écrire deux fonctions de résolution du problème SAT, l'une par recherche exhaustive et l'autre par *backtrack*. Pour cela, on utilisera la représentation informatique des formules vue en cours (*cf* Représentation des formules). Les formules de test sont fournies dans des fichiers `.cnf`.

Rappel sur le problème. Le problème SAT prend en entrée une formule sous forme normale conjonctive (*formule* CNF en abrégé), et renvoie une affectation des variables qui satisfait la formule, ou répond que la formule n'est pas satisfiable. Une formule CNF est une *conjonction de clauses*, chaque clause étant une *disjonction de littéraux* et un littéral étant soit une variable, soit sa négation. Par exemple, $\varphi(x_1, x_2, x_3) = (\neg x_1 \vee x_2) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge \neg x_2$ est une formule CNF dont les trois clauses sont $(\neg x_1 \vee x_2)$, $(x_1 \vee x_2 \vee \neg x_3)$ et $\neg x_2$.

Représentation des formules. On représente une formule CNF en Python par une liste de listes d'entiers. Une formule φ de k clauses est représentée par un liste de taille k. Chaque clause est elle-même représentée par un liste de littéraux. On représente le littéral x_i par l'entier i et le littéral $\neg x_i$ par $-i$. Ainsi, la formule φ ci-dessus est représentée par `F = [[-1,2],[1,2,-3],[-2]]`.

Format de fichier. Le format de fichier « DIMACS » est le format standard pour représenter les formules SAT. C'est un fichier texte très proche de la représentation utilisée en Python, avec extension `.cnf` et constitué des lignes suivantes :

- une ligne de la forme `p cnf 3 5` qui signifie que la formule possède 3 variables et 5 clauses ;
- pour chaque clause, une ligne de la forme `1 2 -3 0` pour la clause $x_1 \vee x_2 \vee \neg x_3$, qui liste les littéraux de la clause et finit par un `0` ;
- des *lignes de commentaires*, n'importe où dans le fichier, commençant par le caractère `c` et qui peuvent (et doivent) être ignorées.

Par exemple, la formule de l'exemple peut être représentée par le fichier suivant.

```
c Formule d'exemple, avec 3 variables et 3 clauses
p cnf 3 3
-1 2 0
1 2 -3 0
-2 0
```

Fichiers de test. Les fichiers de tests sont contenus dans l'archive `cnf.tar.gz`. La formule la plus simple est contenue dans le fichier `simple_v3_c2.cnf`. Les fichiers `quinn.cnf` et `hole6.cnf` correspondent à des formules plus compliquées. Enfin, les fichiers `random-`*i*`-sat.cnf` et `random-`*i*`-unsat.cnf` contiennent des formules générées aléatoirement, à i variables, satisfiables ou non satisfiables selon leur nom.

À vous de jouer ! Avant tout, téléchargez le code à trou `TP2-exo1.py`.

1. Le fichier `TP2-exo1.py` contient les deux fonctions `parseur(nom)` et `affiche(F)` qui permettent respectivement de récupérer une formule SAT depuis un fichier .cnf sous forme de tableau et de l'afficher sous forme plus lisible sur le terminale. Observez le code de ces fonctions puis testez les avec les fichiers `simple_v3_c2.cnf` puis `quinn.cnf`.
2. On implante dans cette question la résolution de SAT par recherche exhaustive. On rappelle qu'on représente les booléens par des entiers 1 (pour VRAI) ou -1 (pour FAUX) et qu'une affectation de n variables est une liste de taille n.
    1. Écrire une fonction `est_valide(F, A)` qui prend en entrée une formule `F` et une affectation `A` et teste si l'affectation `A` satisfait la formule `F`. ***Tester*** *avec la formule contenue dans* `simple_v3_c2.cnf` *et les affectations* (VRAI, VRAI, FAUX) *et* (FAUX, FAUX, VRAI).
    2. Écrire une fonction `aff_suivante(A)` qui prend en entrée une affectation `A` et calcule l'affectation suivante, avec l'algorithme vu en cours. *Si* `A` *est la dernière affectation, l'algorithme renvoie* `None`. ***Tester*** *en affichant, dans l'ordre, toutes les affectations à* 4 *variables.*

3. Écrire une fonction `sat_exhau(F, n)` qui prend en entrée une formule `F` à `n` variables et renvoie une affectation `A` (de longueur `n`) qui satisfait `F`, s'il en existe une, et `None` sinon, par recherche exhaustive. On utilisera les fonctions `est_valide(F, A)` et `aff_suivante(A)` comme indiqué dans le cours. ***Tester*** *avec la formule du fichier* `simple_v3_c2.cnf`*, puis avec celles des fichiers* `random-...cnf` *pour différentes valeurs de* i*. Jusqu'à quelle valeur de* i *pouvez-vous trouver une solution pour une formule satisfiable en environ 1 seconde ? Et pour les formules insatisfiables ?*

3. On implante dans cette question la résolution de SAT par *backtrack*.

1. Écrire une fonction `elimination(F, n, b)` qui prend en entrée une formule `F` à `n` variables et un booléen `b` (ici, un entier ± 1), et renvoie la formule à $n-1$ variables obtenue en *éliminant* la variable x_n en lui affectant la valeur `b`. ***Tester*** *que votre fonction est correcte sur de petits exemples.*
2. Écrire une fonction `sat_backtrack(F, n)` qui prend en entrée une formule `F` à `n` variables et renvoie une affectation `A` (de longueur `n`) qui satisfait `F` s'il en existe une, `None` sinon, par *backtrack*. ***Tester*** *avec la formule du fichier* `simple_v3_c2.cnf`*, puis avec celles des fichiers* `random-...cnf` *pour différentes valeurs de* i*. Jusqu'à quelle valeur de* i *pouvez-vous trouver une solution pour une formule satisfiable en environ 1 seconde ? Et pour les formules insatisfiables ? La formule contenue dans le fichier* `hole6.cnf` *est-elle satisfiable ?*

Exercice 2. *Sudoku*

Le but de l'exercice est d'écrire un algorithme de résolution de grilles de Sudoku *généralisées*, c'est-à-dire des grilles de Sudoku $n^2 \times n^2$ pour n'importe quel n.

Rappel des règles. Une grille complète de Sudoku $n^2 \times n^2$ contient des entiers entre 1 et n^2. Pour qu'elle soit valide, chaque ligne ne doit contenir que des entiers distincts, ainsi que chaque colonne et chaque *zone* (on découpe la grille $n^2 \times n^2$ en n^2 zones de taille $n \times n$). L'entrée est une grille non complète (certaines cases sont vides). La sortie est la grille complétée ou `None` si la grille n'a aucune solution.

Représentation informatique. On représente la grille par une liste G de longueur n^4. La case $G_{[u]}$ de G représente la case (i, j) de la grille, où i et j sont le quotient et le reste dans la division euclidienne de u par n^2. Inversement, la case (i, j) de la grille est stockée dans la case $G_{[in^2+j]}$. Les cases contiennent des entiers entre 1 et n^2, ou 0 pour représenter une case vide.

Format de fichier. On représente une grille de Sudoku dans un fichier texte de la manière suivante. La première ligne du fichier contient l'entier n ($n = 3$ pour un Sudoku *standard* 9×9). Les n^2 lignes suivantes du fichier contiennent chacun une ligne de la grille de Sudoku. Une ligne est représentée par la liste des entiers qu'elle contient, séparés par des espace, et peut contenir des 0 pour représenter les cases vides. Par exemple, la ligne `0 0 3 0 2 0 6 0 0` représente une ligne avec un 3 en 3ème colonne, un 2 en 5ème colonne et un 6 en 7ème colonne.

Fichiers de test. L'archive `sudokus.tar.gz` contient des grilles 9×9 (classées par difficulté). Pour les programmes les plus efficaces, cinq grilles 16×16 et une grille 25×25 sont aussi fournies.

1. Lecture, écriture, affichage.

1. Écrire une fonction `lecture_sudoku(nom)` qui prend en entrée un nom de fichier, et renvoie le couple `(G,n)` où `G` est la liste de longueur n^4 correspondant à la grille de Sudoku contenue dans le fichier. *Inspirez vous des fonctions parseur() et clause() de l'exercice précédent.*
2. Écrire une fonction `affiche_sudoku(G, n)` qui prend en entrée une liste `G` de dimension n^4 et affiche à l'écran la grille de Sudoku correspondante, ligne par ligne, avec des espaces entre les valeurs. Représenter les cases vides par le caractère `_`.

2. Résolution par *backtrack*.

1. Écrire une fonction `zone(n, u)` qui, étant donné `n` et un indice `u` (entre 0 et n^4-1), renvoie la liste des indices de toutes les cases qui sont dans la même zone que `u`. On pourra utiliser la fonction `divmod(`$u, n*n$`)` qui retourne le couple formé par le quotient et le reste de la division entière de u par $n*n$. *Par exemple,* `zone(3,4)` *doit renvoyer* `[3, 4, 5, 12, 13, 14, 21, 22, 23]` *et* `zone(4, 50)` *doit renvoyer* `[0, 1, 2, 3, 16, 17, 18, 19, 32, 33, 34, 35, 48, 49, 50, 51]`.
2. Écrire une fonction `valide(G, n, u, x)` qui prend en entrée une grille `G` de dimensions $n^2 \times n^2$, un indice $u < n^4$ et une valeur x, et renvoie `True` si ajouter la valeur x en case d'indice u ne crée aucun conflit, et `False` sinon. ***Tester*** *sur quelques exemples.*

3. Écrire une fonction `sudoku(G, n, u = 0)` qui prend en entrée une grille `G` de dimensions $n^2 \times n^2$ et un indice (optionnel) $u < n^4$, et complète la grille `G` et renvoie `True` si la grille possède une solution, et ne modifie pas la grille et renvoie `False` sinon, en utilisant un algorithme de type *backtrack*. ***Tester** sur différentes grilles. Jusqu'à quelle dimension de grille pouvez-vous résoudre le problème en environ 1 seconde ?*

Exercice 3. *Cryptarithme*

Un *cryptarithme* est un casse-tête logique qui consiste en une équation mathématique où les lettres remplacent des chiffres à trouver. Par exemple résoudre le cryptarithme `OASIS + SOLEIL = MIRAGE` consiste à assigner des chiffres entre 0 et 9 à chacune des lettres afin que l'équation soit valide. Ici, il faut trouver $73858 + 876456 = 950314$, c'est-à-dire $O = 7$, $A = 3$, etc.

1. On représente des permutations de $\{0, \dots, 9\}$ par des tableaux de 10 entiers tous distincts. Implanter l'algorithme du cours pour passer d'une permutation à la suivante. *Pour tester, afficher par exemple les* 10 *premières permutations.*
2. Écrire une fonction `dico(a,b,c)` qui prend en entrée trois mots et renvoie un dictionnaire Python où chaque lettre présente dans l'un des trois mots `a`, `b` ou `c` est associée à un entier entre 0 et 9. Pour initaliser un dictionnaire en Python, on peut utiliser $d = \{\}$ ou $d =$`dico()`. *Attention : deux lettres distinctes doivent être associées à deux entiers distincts. Par exemple, appliquée aux trois mots* `'OASIS'`, `'SOLEIL'` *et* `'MIRAGE'`*, la fonction peut renvoyer le dictionnaire* `{'O':1, 'A':2, 'S':3, 'I':4, 'L':5, 'E':6, 'M:7', 'R':8, 'G':9}`.
3. Écrire une fonction `valeur(m, D, p)` où `m` est un mot, `D` un dictionnaire tel que celui renvoyé à la question précédente et `p` une permutation, et renvoie la valeur obtenue à partir de `m` en remplaçant chaque lettre `x` par l'entier `p[D[x]]`.
4. Écrire une fonction `cryptarithme(a, b, c)` qui résout un cryptarithme `a + b = c`. *La fonction renvoie un dictionnaire qui associe à chaque lettre un entier entre* 0 *et* 9.

   Exemples de cryptarithmes à tester :

```
COCA + COLA = PEPSI              LECON + ELEVE = DEVOIR
FRERE + SOEUR = BASTON           ARGENT + ARGENT = MALHEUR
HUIT + HUIT = SEIZE              NEUF + DEUX = ONZE
EPOUX + EPOUX = COUPLE           CHEVAL + VACHE = OISEAU
EPOUX + EPOUSE = COUPLE          EPOUSE + EPOUSE = COUPLE
```

Exercice 4. *Huit reines*

Le problème des huit reines demande s'il est possible de placer 8 reines sur un échiquier 8×8, sans qu'elles se menacent l'une l'autre : deux reines se menacent si elle sont sur la même ligne, la même colonne, ou la même diagonale (ou anti-diagonale). C'est en effet possible, et il y a même plusieurs solutions. On généralise le problème au cas des n reines : on cherche à placer n reines sur un échiquier $n \times n$, avec toujours les mêmes conditions.

On remarque que chaque ligne et chaque colonne contient exactement une reine. On représente une solution par un tableau `Q` de taille n, tel que `Q[i]` vaut `j` s'il y a une reine en case (i, j). On utilise `Q[i] = -1` s'il n'y a pas (encore) de reine sur la ligne i.

1. Écrire une fonction `affichage_reine(Q)` qui affiche le damier, avec le symbole `_` pour les cases vides et `x` pour les cases qui contiennent une reine. *Par exemple, si* `Q = [2,0,3,1]`*, l'affichage est*

```
_ _ x _
x _ _ _
_ _ _ x
_ x _ _
```

2. Écrire une fonction `reine_possible(Q, r, j)` qui prend en entrée le tableau `Q` dont les `r` premières lignes sont remplies (sans conflit entre les reines), et qui répond `True` s'il est possible de placer une reine en position (r, j), et `False` sinon. *Par exemple, si* $Q = [1, 3, -1, -1]$*,* `reine_possible(Q, 2, 0)` *renvoie* `True` *mais* `reine_possible(Q, 2, 2)` *renvoie* `False` *car une reine en case* $(2, 2)$ *serait en conflit avec celle en case* $(1, 3)$.
3. Écrire une fonction de type *backtrack* `placement_reines(Q, r=0)` qui prend en entrée le tableau `Q` dont les `r` premières reines sont placées (sans conflit) et renvoie le tableau `Q` totalement rempli (si c'est possible) et `None` sinon. *Par exemple, sur l'entrée* `Q = [2, 0, 3, -1]` *et* `r=3`*, la fonction doit renvoyer* `[2,0,3,1]` *alors que si* `Q = [0, 3, 1, -1]`*, elle doit renvoyer* `None`.
4. Écrire une fonction `reines(n)` qui prend en entrée `n` et renvoie, si possible, un tableau `Q` valide. Vérifier qu'il n'existe pas de solution pour $n = 2$ et $n = 3$. Quelle taille de damier pouvez-vous traiter en environ 5 secondes ?