BUT1 – S.A.E. S1-02
ETUDIER LA SATISFIABILITÉ D'UN ENSEMBLE DE CONTRAINTES

IUT DE NANTES – DÉPARTEMENT D'INFORMATIQUE

Contact : François Simonneau, Email : francois.simonneau@univ-nantes.fr

L'objectif de ce projet est de se familiariser avec des techniques de base de test de satisfiabilité booléenne via l'implémentation d'un solveur SAT et d'utiliser la résolution SAT pour résoudre des Sudoku.

Le projet se découpe ainsi en deux parties : une partie solveur SAT pour des formules en forme normale conjonctive (notée cnf) et une partie modélisation en cnf de grilles de Sudoku :

1. La première partie se concentre sur la modélisation de formules propositionnelles, leur évaluation et la résolution du problème par différentes méthodes de degrés d'efficacité croissants (la résolution d'un problème de Sudoku classique exigera la mise en place d'un solveur sufisamment efficace).
2. La seconde partie est une modélisation des contraintes apparaissant dans une grille de Sudoku pour que la résolution de celle-ci soit réalisable avec le solveur mis en place en première partie

L'évaluation sera individualisée et elle se clôturera avant le début des vacances de Noël avec différentes échéances (et barème approximatif indicatif) :

— Evaluation sur mise sous forme normale conjonctive de formules propositionnelles, utilisation d'arbres pour la résolution avec simplification de formules en fonctions d'une hypothèse émise sur une variable (lors de l'évaluation de la ressource Maths discrètes avant les vacances d'automne - 5 points) ;

— Evaluation tp avec fonctions décrites dans ce sujet à savoir implémenter lors de l'évaluation surveillée et en temps limité (Fin Novembre - Début Décembre a priori - 15 points, elle se fera sur deux demi-journées séparées et lors de la deuxième séance d'évaluation vous pourrez repartir du code établi à l'issue de la première séance) ;

Partie 1 – Résolution de problème SAT

1.1. **Structure de données pour représenter une formule conjonctive**. Les formules cnf ayant une forme très régulière on va pouvoir les représenter par une liste de clauses. Les clauses sont elles-mêmes représentées comme des listes de littéraux. On considère la formule :

$$(p \vee q \vee \neg r) \wedge (\neg p \vee \neg q) \wedge (\neg p \vee r) \wedge (\neg p \vee q \vee \neg r)$$

On peut y retrouver :

— 4 clauses : $(p \vee q \vee \neg r)$, $(\neg p \vee \neg q)$, $(\neg p \vee r)$ et $(\neg p \vee q \vee \neg r)$

— 3 variables : p, q et r

— 6 littéraux : chaque variable peut intervenir « telle quelle »ou via sa négation, soit deux littéraux différents (ex : p et $\neg p$)

A chaque variable, on associe un numéro (numérotation initialisée à 1) et lorsqu'une variable apparaît telle quelle dans une clause, on le représente en faisant apparaître son numéro dans la liste, et quand elle apparaît via sa négation on fait apparaître l'opposé de ce numéro. Ainsi pour la formule ci-dessus :

— La clause $(p \vee q \vee \neg r)$ sera modélisée par la liste `[1,2,-3]`

— La formule sera modélisée par la liste `[[1,2,-3],[-1,-2],[-1,3],[-1,2,-3]]`

Pour information cette modélisation s'inspire du format de fichier `Dimacs` régulièrement utilisé par les solveurs SAT.

1.2. **Valuations associées à l'ensemble des variables**. A chaque variable on peut associer les valeurs suivantes :

— `True`

— `False`

— `None` tant qu'on n'a pas affecté une valeur à cette variable

Le problème SAT peut se résumer en deux objectifs majeurs :

— Déterminer si une formule est satisfiable (on renverra `True`) ou insatisfiable (on renverra `False`)

— Dans le cas où elle est satisfiable, renvoyer une valuation permettant de vérifier la formule (i.e. une liste donnant une affectation à l'ensemble des variables intervenant dans la formule)

Pour résoudre le problème SAT nous allons parcourir un ensemble de valuations envisageables. On représentera classiquement une valuation par l'argument `list_var` donnant une valeur affectée à l'ensemble des variables. Ainsi pour l'exemple précédent on peut envisager d'initialiser cette liste à :

— `[None,None,None,None]` si aucune contrainte initiale n'est formulée

— `[True,None,None,None]` si une contrainte supplémentaire était formulée ou en commençant l'initialisation des tests à effectuer par cette valuation : en l'occurrence l'attribution de la valeur `True` à la variable p et auquel cas on verra qu'on pourrait aussi changer l'expression de la formule pour étudier de manière équivalente la satisfiabilité d'une formule simplifiée

Attention : Il ne faut pas confondre le numéro associé à une variable (qui permet dans une formule de définir le littéral associé en fonction de la valeur logique attendue pour cette variable) avec la position de cette variable dans une liste. Ainsi dans l'exemple ci-dessus la variable p est numéroté à 1 ce qui permet de représenter les littéraux p et $\neg p$ par 1 et -1 dans les clauses de la formules. Par contre dans les listes énumérant les valeurs logiques attribuées aux variables ou les changements opérés sur celles-ci nous renseignerons les positions de ces variables. Ainsi dans la `list_var` donnant une valuation en cours de construction lorsqu'on affecte la valeur `True` on exploite la position de cette variable soit `list_var[0]=True` et si on veut ajouter ce changement à une liste de changements enregistrés on le fera également via cette position : `list_chgmts.append([0,True])`.

1.3. **Evaluation d'une formule**. La valeur d'une clause peut-être déterminée comme suit :

- `True` si elle possède un littéral dont la valeur est `True` ;
- `False` si tous ses littéraux sont évalués à `Faux` ou si la clause est vide ;
- `None` dans les autres cas.

Attention par contre une formule vide doit se voir attribuer la valeur logique `True` pour assurer la compatibilité avec les algorithmes vus utilisant des méthodes de simplifications de formules.

Vous devrez rédiger les fonctions suivantes.

Q. 1.3.1. Une fonction `evaluer_clause(clause,list_var)` qui renvoie l'évaluation de la clause passée en argument pour la valuation passée en argument (`list_var`)

Q. 1.3.2. Une fonction `evaluer_cnf(formule,list_var)` qui renvoie l'évaluation de la formule passée en argument pour la valuation passée en argument (`list_var`)

1.4. **Résolution par étude de toutes les valuations**.

Q. 1.4.1. Une fonction `determine_valuations(list_var)` qui renvoie une liste avec l'ensemble des valuations à tester pour la liste de variables passée en argument. Chaque valuation doit correspondre à une liste donnant les valeurs logiques assignées à chaque variable propositionnelle (attention vous veillerez à respecter les valeurs déjà assignées dans la liste de variables passée en argument). Ainsi par exemple pour une liste de 4 variables avec deux contraintes connues sur la première (`True`) et 3ème variable (`False`), l'appel `determine_valuations([True,None,False,None])` renverra :

`[[True,False,False,False],[True,False,False,True],`

`[True,True,False,False],[True,True,False,True]]` (peu importe l'ordre entre les valuations dans la liste renvoyée). Pour réaliser ceci on peut envisager différentes pistes :

a. Par exemple initialiser une liste de résultats avec la liste passée en arguments et à chaque tour d'une boucle identifier une variable non contrainte et pour chaque liste apparaissant dans la liste de résultats précédente, la remplacer par deux listes où la valeur non contrainte serait remplacée par les valeurs `True` et `False`. Ainsi à partir de la liste `[True,None,False,None]`, on obtiendrait progressivement :

- `[[True,None,False,None]]`
- `[[True,True,False,None],[True,False,False,None]]`
- `[[True,True,False,True],[True,True,False,False],`
  `[True,False,False,True],[True,False,False,False]]`

b. Ou encore, en partant du nombre de variables sans contraintes connues, on peut déterminer le nombre de listes à renvoyer et associer un numéro à chacune d'elle et exploiter l'écriture en binaire de ce numéro avant de transformer ces valeurs en `True` et `False` et d'y placer les contraintes déjà connues aux bons emplacements

Q. 1.4.2. Une fonction `resol_sat_force_brute(formule,list_var)` qui, à partir de l'évaluation de l'ensemble des valuations associables à `list_var`, renvoie un booléen associé à la satisfiabilité de la formule et le cas échéant une valuation solution.

1.5. **Résolution par parcours d'un arbre**. Attention on met en garde sur le fait que les fonctions à réaliser par parcours d'arbre ne doivent pas utiliser `resol_sat_force_brute`, ceci ne permettant pas de renvoyer systématiquement le bon résultat prévu avec l'ordre de parcours vu précédemment sur papier en TD, et selon aussi l'endroit duquel on part dans le parcours de l'arbre. Ce dernier aspect dépend de `list_chgmts`, mais pour une recherche de solution globale partant de la racine on exécute la fonction avec une `list_chgmts` passée en argument correspondant à une liste vide.

Vous devrez, pour la mise en place de cette méthode, réaliser les fonctions suivantes :

Q. 1.5.1. Une fonction `progress(list_var,list_chgmts)` qui permet de descendre dans l'arbre jusqu'à un premier changement à apporter à la liste de variables (affectation de la valeur `True`). La fonction renvoie alors la nouvelle valuation en cours `list_var` et la nouvelle liste des changements apportés à `list_var` depuis le lancement de la résolution de la formule. Cette liste des changements renvoyée correspond donc à la liste des changements précédents à laquelle on rajoute une liste correspondant au changement apporté. Ainsi par exemple si à partir d'une liste de changements initiale `[0,True]` on affecte la valeur `True` à la variable en position 2 (dans le cas où la variable en position 1 serait déjà contrainte), on rajoutera la liste `[2,True]` et on renverra la liste `[[0,True],[2,True]]` ;

Q. 1.5.2. Une fonction `retour(list_var,list_chgmts)` qui va permettre de remonter dans l'arbre en annulant les effets des changements précédemment effectués et en supprimant leurs enregistrements, jusqu'à arriver à un branchement où on puisse effectuer un nouveau changement en redescendant vers la droite dans l'arbre (affectation de la valeur `False`), auquel cas on renverra la nouvelle liste des valeurs affectées aux variables et la nouvelle liste des changements en cours. Une attention sera à apporter au cas où la liste des changements serait vide ;

Q. 1.5.3. Une fonction `resol_parcours_arbre(formule_init,list_var,list_chgmts)` qui va évaluer la formule donnée initialement avec la liste des valuations en cours et va, en fonction du résultat, décider éventuellement de la progression du parcours dans l'arbre ou du retour sur des hypothèses émises précédemment. Avec une approche récursive elle doit in fine renvoyer deux éléments :

— Un booléen précisant si la formule est satisfiable ou non sur la suite du parcours dans l'arbre

— Une liste : la valuation solution si la formule est satisfiable ou la liste vide si elle ne l'est pas.

1.6. **Résolution par parcours d'un arbre et simplification de formule**. On pourra, en rappel de l'exemple présenté en cours, s'inspirer de la video suivante de Pascal Ortiz pour comprendre le principe de l'algorithme : https://youtu.be/lRQIk8Ly594 Vous devrez, pour sa mise en place, réaliser les fonctions suivantes :

Q. 1.6.1. La fonction `enlever_litt_for(formule,litteral)` va simplifier l'expression en fonction de la façon dont le **littéral** ou son opposé apparaissent dans les différentes clauses de la formule. Elle renvoie donc une formule simplifiée.

Q. 1.6.2. La fonction `init_formule_simpl_for(formule_init,list_var)` va permettre de simplifier une formule initiale dans laquelle on n'aurait pas tenu compte de l'ensemble des valeurs déjà attribuées à `list_var`;

Q. 1.6.3. La fonction `retablir_for(formule_init,list_chgmts)` va prendre en arguments une formule initiale à simplifier selon une liste de changements sur un ensemble de variables et va renvoyer la formule simplifiée en tenant compte de l'ensemble de ces changements;

Les fonctions suivantes s'appuieront sur les fonctions précédentes pour la simplification d'une formule logique ou le retour en arrière sur ces simplifications :

Q. 1.6.4. La fonction `progress_simpl_for(formule,list_var,list_chgmts)` permet de descendre dans l'arbre jusqu'à un premier changement à apporter à la liste de variables (affectation de la valeur `True`). La fonction renvoie alors la formule logique simplifiée, la nouvelle valuation en cours `list_var` et la nouvelle liste des changements apportés à `list_var` depuis le lancement de la résolution de la formule (précisions dans la description de `progress(list_var,list_chgmts)`);

Q. 1.6.5. Une fonction `retour_simpl_for(formule_init,list_var,list_chgmts)` qui va permettre de remonter dans l'arbre en annulant les effets des changements précédemment effectués et en supprimant leurs enregistrements, jusqu'à arriver à un branchement où on puisse effectuer un nouveau changement en redescendant vers la droite dans l'arbre (affectation de la valeur `False`), auquel cas on renverra la nouvelle formule à prendre en compte (établie à partir de la formule initiale définie au début de la résolution et de la nouvelle liste des changements à prendre en compte), la nouvelle liste des valeurs affectées aux variables, la nouvelle liste des changements en cours. Une attention sera à apporter au cas où la liste des changements serait vide.

Une fonction fera la synthèse des précédentes pour résoudre le problème :

Q. 1.6.6. La fonction principale

`resol_parcours_arbre_simpl_for(formule_init,formule,list_var,list_chgmts)`

va évaluer la formule en cours avec la liste des valuations en cours et va en fonction du résultat décider éventuellement la progression du parcours dans l'arbre ou le retour sur des hypothèses émises précédemment. Avec une approche récursive elle doit in fine renvoyer deux éléments :

- Un booléen précisant si la formule est satisfiable ou non sur la suite du parcours dans l'arbre
- Une liste : la valuation solution si la liste est satisfiable ou la liste vide si elle ne l'est pas.

On considère par ailleurs que la `formule` a bien été initialisée avec `init_formule_simpl_for` avec ce qui figure dans `list_var`. L'exploitation de `resol_parcours_arbre_simpl_for` peut se faire dans le cadre d'une recherche globale de solution, c'est-à-dire en partant de la racine de l'arbre (avec `formule_init` identique à `formule` et avec une `list_chgmts` vide) ou dans le cadre d'une recherche à partir d'un point précis du parcours (en fonction de `list_chgmts`).

Attention : Lorsqu'on effectue la copie d'une liste, un changement effectué sur la copie n'affectera pas la liste initiale à condition que les éléments eux-mêmes de cette liste ne soient pas mutables. Dans le cas contraire, comme pour une liste de listes il faudra effectuer une copie en profondeur de cette liste. La bibliothèque `copy` propose une fonction `deepcopy` effectuant ceci.

1.7. **Résolution avec algorithme DPLL.** On pourra, en rappel de l'exemple présenté en cours, s'inspirer de la video suivante de Pascal Ortiz, sans toutefois prendre en compte l'heuristique MOMS (par défaut on effectuera le parcours de l'arbre dans l'ordre de numérotation des variables) et il est plutôt recommandé de commencer à étudier les clauses unitaires avant les litteraux purs : https://youtu.be/GYXc285r3nc

Des fonctions réalisées avec la solution précédente devront être revues pour notamment tenir compte des affectations de valeurs à des variables sur lesquelles on ne souhaite pas envisager d'alternative en cas de backtracking (cas des clauses unitaires et littéraux purs). Ainsi un argument `list_sans_retour` contiendra la position des variables concernées. Cet argument devra être ajouté et pris en compte pour construire les fonctions suivantes :

Q. 1.7.1. `progress_simpl_for_dpll`

Q. 1.7.2. `retour_simpl_for_dpll`

Q. 1.7.3. `resol_parcours_arbre_simpl_for_dpll`

Partie 2 – Modélisation d'une grille de Sudoku

2.1. **Présentation générale.** Le problème de Sudoku consiste à placer des nombres sur une grille de dimension $n^2 \times n^2$ (pour le sudoku classique $n = 3$). Certaines cases ont déjà une valeur qui leur est assignée (soit un nombre compris entre 1 et n^2 et les autres sont à remplir lors de la résolution (la valeur 0 leur est initialement attribuée). La grille est subdivisée en régions disjointes qui sont des sous-grilles de taille $n \times n$. On suppose que les lignes et les colonnes sont numérotées en partant du haut et de la gauche. Compléter une grille de Sudoku se fait en respectant certaines contraintes. On va les exprimer de la manière suivante pour construire notre formule logique (d'autres façons d'exprimer ces contraintes menant vers des formules plus synthétiques seraient envisageables mais celle-ci permettra d'avoir des résultats satisfaisants dans la recherche de solutions)

— Chaque nombre doit être présent une et une seule fois sur chaque ligne (C_1).

— Chaque nombre doit être présent une et une seule fois sur chaque colonne (C_2)

— Chaque nombre doit être présent une et une seule dans chaque région (C_3)

— Les nombres placés sont tous compris entre 1 et n^2 inclus.

On ajoutera par ailleurs la contrainte évidente suivante : plusieurs nombres ne peuvent être saisis dans une même case (C_4).

Afin de coder le problème de Sudoku au format cnf on introduit n^2 variables booléennes par case de la grille. On note $p^v_{i,j}$ la v^{me} variable de la case située à la ligne i et la colonne j. Cette variable représente alors le fait que le nombre v doit être ou non affecté à cette case pour résoudre le problème (elle prend donc par exemple la valeur `True` si on affecte le nombre v à cette case). On suppose que l'on numérote l'ensemble de ces variables en partant de la case en haut à gauche et en numérotant les variables associées à une même case consécutivement puis en passant à la case directement à droite sur une même ligne et en reprenant à la première case la plus à gauche de la ligne suivante. Cela se traduit par les règles de numérotations suivantes :

— $num(p^1_{0,0}) = 1$

— $num(p^v_{i,j}) = num(p^{v-1}_{i,j}) + 1$ si $v \geq 2$

— $num(p^1_{i,j}) = num(p^{n^2}_{i,j-1}) + 1$ si $j \geq 1$

— $num(p^1_{i,0}) = num(p^{n^2}_{i-1,n^2-1}) + 1$ si $i \geq 1$.

2.2. **Fonctions à réaliser.**

Q. 2.4. Créer une fonction `for_conj_sudoku(n)` prenant en argument la dimension de la grille voulue (nombre de ligne et colonne par région) et renvoyant la formule normale conjonctive attendue permettant le respect des contraintes C_1 à C_4. Les contraintes C_1 et C_2 se traduiront chacune par n^4 clauses de n^2 littéraux (chaque nombre doit être présent au moins une fois) et $\frac{n^6(n^2-1)}{2}$ clauses binaires (mais pas de doublon). En évitant des doublons avec des clauses déjà précédemment établies, la contrainte C_3 se traduira par n^4 clauses de n^2 littéraux et $\frac{n^6(n-1)^2}{2}$ clauses binaires. La contrainte C_4 se traduira par $\frac{n^6(n^2-1)}{2}$ clauses binaires

Q. 2.5. Créer une fonction `creer_grille_init(list_grille_coord_connues,n)` prenant en arguments une liste de listes(contenant les coordonnées des cases renseignées dans la grille et le nombre correspondant) et un nombre n spécifiant la taille de la grille voulue et renvoyant une liste de taille n^4 attribuant la valeur 0 à toutes les cases pour lesquelles la valeur est inconnue et la valeur figurant dans la liste en argument pour les autres. Ainsi avec `[[1,2,1],[2,1,4],[2,2,2],[3,3,2],[4,2,3]]` et `n=2` on renverrait la liste : `[0, 1, 0, 0, 4, 2, 0, 0, 0, 0, 2, 0, 0, 3, 0, 0]`

Q. 2.6. Créer une fonction `afficher_grille(list_grille_complete,n)` qui, à partir d'une liste au format correspondant à une liste renvoyée par la fonction précédente, permettra l'affichage de la grille sous sa forme habituelle (on pourra par exemple simplement utiliser la fonction `reshape` de la bibliothèque `numpy`. Ainsi à partir de l'exemple précédent on obtiendrait la grille (la mise en forme avec les lignes n'est pas attendue) :

0	1	0	0
4	2	0	0
0	0	2	0
0	3	0	0

Q. 2.7. Créer une fonction `init_list_var(list_grille_complete,n)` permettant de renvoyer une initialisation de la liste de valuations `list_var` en tenant compte des valeurs déjà renseignées dans `list_grille_complete`.

Q. 2.8. Créer une fonction `creer_grille_final(list_var,n)` permettant, à partir d'une valuation attribuée à l'ensemble des variables associées à un problème de sudoku, de renvoyer la grille avec le même format que celui utilisé dans les renvois de la fonction `creer_grille_init`.

Q. 2.9. Tester la résolution des grilles de Sudoku proposées avec les solutions développées dans la première partie.

Exemple pour n=2 : n=2 signifie ici que la grille est construite comme une grille de 4 lignes et 4 colonnes permettant de délimiter 4 régions (carrés de 2 lignes et 2 colonnes). Chaque case de la grille peut être renseignée avec un nombre variant de 1 à 4. Chaque case donne alors lieu à 4 variables logiques auxquelles on affecte les valeurs True ou False selon qu'un nombre est placé dans cette case ou non (ou None tant que la situation est indéterminée). Il y a 16 cases dans la grille donc 64 variables au total. Ces variables sont numérotées dans le même ordre que l'ordre de lecture (de la gauche vers la droite à partir de la première ligne puis on reprend à la ligne suivante de la gauche vers la droite). Ainsi la première case a sa valeur associée modélisée par la variable 1 qui correspond au fait d'y mettre un 1, la variable 2 qui correspond au fait d'y mettre un 2... La deuxième case (1ère ligne, 2ème colonne) est modélisée par la variable 5 qui correspond au fait d'y mettre un 1, la variable 6 qui correspond au fait d'y mettre un 2... La 5ème case (2ème ligne, 1ère colonne) est modélisée par la variable 17 qui correspond au fait d'y mettre un 1, la variable 18 qui correspond au fait d'y mettre un 2.... Une fois qu'on a compris ceci il faut traduire les différentes règles énoncées, par exemple : C1 évoque le fait que la valeur 1 doit être quelque part sur la ligne 1 c'est à dire que la variable 1 doit être égale à True ou la variable 5 ou la variable 9 ou la variable 13 cela donne lieu à une clause [1,5,9,13] mais elle ne doit pas apparaitre deux fois donc par exemple non(et(1,5)) soit ou(non(1),non(5)) c'est à dire une clause [-1,-5].