DS4 : type CCINP

Ce sujet est composé d'un exercice sur la concurrence, d'un exercice de logique et d'un problème portant sur le problème MaxSat.

I Exercice : Concurrence

Nous allons considérer dans cet exercice un programme multithreadé.

Q. 1 Que risque-t-il de se passer si plusieurs fils (threads) incrémentent la même variable.

Q. 2 Proposer une solution permettant de protéger l'accès aux données si deux threads veulent les modifier simultanément, sans recourir aux primitives offertes par le système d'exploitation. On attend une solution algorithmique au programme qui utilise l'attente active.

On suppose maintenant que chaque thread exécute le même code, donné à la suite en C :

```
int nb_threads; //Nombre de threads
pthread_mutex_t *mutex; //Tableau de mutex, partagé entre tous les threads
bool *pas_pris; //Tableau de booléens, partagé entre tous les threads
int id; //L'identifiant du thread : chacun a le sien, entre 0 et nb_threads-1
//Début Partie 1
pthread_mutex_lock(&mutex[id]);
pas_pris[id] = false;
pthread_mutex_unlock(&mutex[id]);
//Fin Partie 1
//Début Boucle
for(int i = 0; i < nb_threads; i+=1){
   while(pas_pris[i]) {}
   pthread_mutex_lock(&mutex[i]);
   pthread_mutex_unlock(&mutex[i]);
}
//Fin Boucle
//Début Partie 2
```

Q. 3 Sous réserve d'une certaine initialisation de `pas_pris` à préciser, démontrer qu'on a l'invariant de boucle "Pour tout $id < i$, le thread id a terminé d'exécuter la partie 1 du code".

Q. 4 En déduire une condition nécessaire pour entrer dans la partie 2 du code.

Q. 5 Montrer qu'il y a toujours un thread en mesure d´avancer dans la boucle et donc qu'il n'y a pas d'interblocage.

Q. 6 En déduire, en mettant en évidence un variant, que la condition proposée à la question 4 est suffisante pour garantir que tous les threads finiront par entrer dans la partie 2 du code au bout d'un certain moment.

II Exercice : Logique

Partie I - Logique

Un footballeur affirme à la presse :

(i). Le jour où je marque un but, je suis content et je fais la fête.

(ii). Le jour où mon équipe gagne, ou bien je suis content, ou bien je fais la fête ou les deux.

(iii). Le jour où mon équipe perd, ou bien je ne suis pas content, ou bien j'ai marqué un but ou les deux.

(iv). Le jour où je ne marque pas et je fais la fête, je suis content.

(v). Aujourd'hui, je ne suis pas content.

Q1. Définir les variables propositionnelles nécessaires à la modélisation de ce problème.

Q2. Modéliser chacune des assertions à l'aide de formules propositionnelles.

Q3. Mettre chacune de ces formules en forme normale conjonctive.

Q4. On souhaite savoir si le joueur a marqué, si son équipe a gagné et s'il a fait la fête. Donner la formule F permettant de répondre à ces questions. À quel problème classique est-on confronté ?

On rappelle l'algorithme de Quine (algorithme 1) : en notant
- C l'ensemble des clauses de F,
- x une variable propositionnelle,
- $C[x \leftarrow \top]$ l'ensemble des clauses obtenues en supprimant de C toutes les clauses contenant x, et en supprimant $\neg x$ de toutes les clauses contenant cette négation,
- $C[x \leftarrow \bot]$ l'ensemble des clauses obtenues en supprimant de C toutes les clauses contenant $\neg x$, et en supprimant x de toutes les clauses contenant cette variable.

l'algorithme s'écrit :

Algorithme 1 : Algorithme de Quine

```
Fonction Quine(C)
Entrées : C l'ensemble des clauses de F.
Sorties : Vrai si F est satisfsiable, Faux sinon.
début
    si C = ∅ alors
        retourner Vrai
    si ⊥ ∈ C alors
        retourner Faux
    Choisir x apparaissant dans une clause de C
    si Quine(C[x ← ⊥]) alors
        retourner Vrai
    sinon
        retourner Quine(C[x ← ⊤])
```

Q5. Appliquer cet algorithme et trouver une valuation qui rende F vraie.

III Problème : MaxSat

Présentation du problème

On rappelle que le problème MaxSat est défini de la manière suivante :
Instance : une formule propositionnelle φ en CNF
Solution : une valuation v (définie sur les variables de φ)
Mesure à maximiser : le nombre de clauses de φ satisfaites par v

Dans tout ce sujet, on manipulera donc des formules propositionnelles sous forme normale conjonctive. On exploite ce format pour représenter une formule propositionnelle φ de manière plus compacte qu'avec un arbre : on représente φ comme la liste de ses clauses, chaque clause étant une liste de littéraux. Les variables propositionnelles sont indexées à partir de 1, et on représente le littéral x_i par l'entier i et le littéral $\neg x_i$ par l'entier $-i$.

On utilisera donc les types OCaml suivants:

```
type litteral = int (* positif pour x_i, négatif pour non x_i *)
type clause = litteral list
type cnf = clause list (* type des formules propositionnelles en CNF *)
```

Q. 1 Écrire une fonction `n-var : cnf -> int` qui prend en argument une formule `phi` et renvoie l'indice maximal `n` d'une variable propositionnelle x_n de `phi`. Si `phi` ne contient aucune variable propositionnelle, on renverra 0.

On représente comme d'habitude une valuation sur les variables $x_1, \ldots, x_n$ par un tableau de booléens `v`. Par souci de simplicité, on ignorera la case d'indice 0 qui contiendra un booléen arbitraire, de sorte que `v.(i)` contienne `true` si $v(x_i) = V$ (c'est à dire $v(x_i)$ est Vrai). Le tableau devra donc être de longueur $n + 1$.

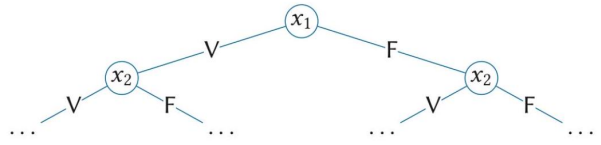
Pour représenter une valuation partielle, on remplit le tableau (de gauche à droite) jusqu'à une certaine case d'indice k, et on retient ce k. Le reste du tableau contient des booléens arbitraires. Ainsi, on n'a pas besoin de créer un nouveau tableau de taille différente à chaque fois qu'on rajoute une variable à notre valuation, mais seulement de continuer à remplir le tableau.

A Solution optimale

Dans cette partie, on va résoudre le problème MaxSat en utilisant la séparation et évaluation (branch and bound).

A. 1 Première version

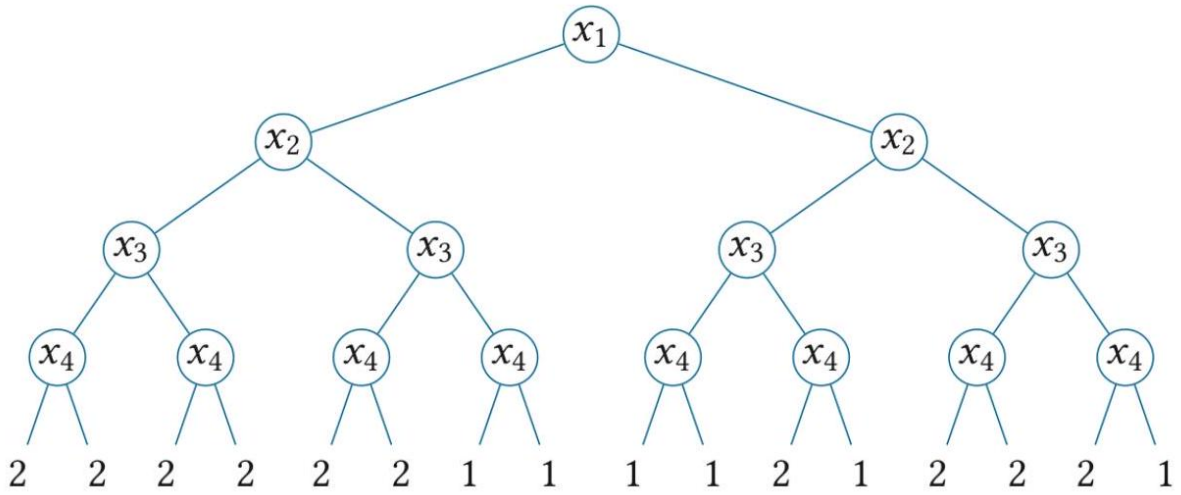
Souvenez-vous de l'algorithme de Quine. Le principe est de représenter la recherche de solution sous la forme du parcours d'un arbre d'exploration. Dans le cas des formules propositionnelles, les solutions recherchées sont des valuations, et les solutions partielles sont des restrictions de valuations à certaines variables. Explorer toutes les valuations possibles revient à tester, pour chaque variable, toutes les valuations qui mettent cette variable à Vrai et toutes les valuations qui la mettent à Faux. On obtient un arbre d'exploration de la forme suivante :



On considèrera à titre d'exemple la formule suivante :

$$\varphi_0 = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_3) \\ \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_3)$$

Dans le cas de la formule φ_0, on a alors l'arbre d'exploration complet suivant, où l'on a indiqué aux feuilles le nombre de clauses qui ne sont pas satisfaites par la valuation obtenue par les choix menant à cette feuille.



Q. 2 En lisant cet arbre, donner une solution optimale pour le problème MaxSat pour la formule φ_0. Combien de clauses satisfait-elle?

Un algorithme naïf consisterait à explorer tout l'arbre et retenir la valuation observée avec le moins de clauses non satisfaites. L'idée de la séparation et évaluation, similaire à celle du retour sur trace et de l'élagage $\alpha - \beta$, est d'arrêter l'exploration d'une branche dès qu'on est sure que la solution partielle qu'elle explore ne pourra pas être complétée en solution qui sera strictement mieux que la meilleure solution qu'on a trouvé jusqu'à maintenant. Dans ce cas, on élague la branche.
Il faut donc d'abord un moyen de jauger une solution partielle, et en particulier de borner la valeur des solutions qu'on peut obtenir en la complétant. En cours, on a proposé une telle évaluation : pour une valuation partielle, on regarde le nombre de clauses qui ne peuvent plus être satisfaites par cette valuation. Par exemple, si v est la valuation qui à x_1 et à x_3 associe Vrai et à x_2 et x_3 associe Faux, alors on sait que la quatrième clause de φ_0 $(\neg x_1 \vee x_2 \vee x_3)$ ne
peut plus être satisfaite, quelle que soit la façon dont on complète v. On sait donc que le nombre de clauses satisfaites par une solution complétant v (quelle que soit la valeur donnée à x_4) est au plus $8 - 1 = 7$ clauses.

Q. 3 Ecrire une fonction `insat_clause : bool array -> int -> clause -> bool` telle que `insat_clause v k c` prend en argument une valuation partielle `v` portant sur les variables x_1 à x_k, et une clause `c` ; et renvoie `true` si la clause `c` ne peut pas être satisfiable par une valuation étendant `v`. Sinon, elle renvoie `false`.
On fera bien attention que si $j > k$, alors la case `v.(j)` du tableau n'est pas la valeur attribuée à x_j, c'est un booléen arbitraire.

Q. 4 En déduire une fonction `insat : bool array -> int -> cnf -> int` telle que `insat v k phi` renvoie le nombre de clauses de `phi` qui ne peuvent pas être satisfaites par une valuation étendant `v`.

Lors de l'exploration de l'arbre, on retient le nombre de clauses insatisfaites par la meilleure valuation trouvée jusqu'à maintenant. On commence par explorer la branche la plus à gauche, correspondant à la valuation suivante :

$$v : \begin{cases} x_1 \mapsto V \\ x_2 \mapsto V \\ x_3 \mapsto V \\ x_4 \mapsto V \end{cases}$$

et on trouve que 2 clauses de φ_0 sont insatisfaites. On retient ce nombre. Plus tard, lorsqu'on examinera la valuation (partielle) suivante:

$$v : \begin{cases} x_1 \mapsto V \\ x_2 \mapsto F \\ x_3 \mapsto V \end{cases}$$

on voit par la fonction `insat v k phi` que cette valuation rend déjà 2 clauses de φ_0 insatisfiables. Il est donc inutile de continuer à explorer cette branche en la complétant par des valeurs de x_4, on sait que les valuations explorées ne feront pas strictement mieux que 2 clauses insatisfaites. Dès qu'on dépasse 2 (de manière large), on peut s'arrêter et élaguer.
Attention, l'exemple donné ici n'est pas nécessairement le premier élagage rencontré lors de l'exécution de l'algorithme!

Q. 5 Faire tourner l'algorithme de séparation et évaluation à la main : sur l'arbre d'exploration donné ci-dessus pour φ_0, indiquer au dessus de chaque noeud le nombre de clauses rendues insatisfiables par la valuation partielle obtenue à ce noeud. Indiquer sur l'arbre les branches élaguées en mettant des croix sur les arêtes de l'arbre où l'on arrête l'exploration.
Indication : la racine représente la valuation vide et son nombre de clauses insatisfiable est 0 .

Q. 6 Écrire une fonction `maxSat : cnf -> bool array` qui prend en argument une formule `phi` et renvoie une valuation `v` optimale pour `phi`, c'est à dire qui maximise le nombre de clauses de `phi` satisfaites par `v`. `maxSat` trouvera `v` par la méthode de séparation et évaluation.

A. 2 Version améliorée

Notre version de séparation et évaluation précédente n'est pas très efficace ni très optimisée : à chaque étage de l'arbre on recalcule pour chaque clause si elle est devenue insatisfiable, sans retenir les clauses insatisfiables qu'on a déjà vu précédemment, ni les clauses déjà satisfaites. Améliorons ça en transformant la formule φ au fur et à mesure qu'on donne des valeurs aux variables, un peu comme pour l'algorithme de Quine.

Q. 7 Écrire une fonction `simplifie : litteral -> cnf -> cnf` qui prend en argument un entier `x` et une formule en CNF `phi`, tel que `x` vaut i si on met la variable x_i à Vrai, et `x` vaut $-i$ si la variable x_i est mise à Faux. Cette fonction renvoie une nouvelle formule `phi'` en CNF telle que:

- Les clauses satisfaites grâce à cette nouvelle valeur sont supprimées de `phi`. (Elles ne compteront pas dans le calcul du nombre de clauses insatisfiables, autant les retirer.)
- Les littéraux rendus faux par cette nouvelle valeur sont supprimées de leurs clauses. Les clauses qui deviennent ou sont vides (i.e. insatisfiables) sont conservées dans la liste de `phi`.

Ainsi, plutôt que de calculer la fonction `insat` comme précédemment, on pourra se contenter de compter le nombre de clauses vides dans `phi` après cette simplification.

Q. 8 Écrire une fonction `clauses\_vides : cnf -> int` qui compte le nombre de clauses vides dans `phi` passée en entrée.

Profitons-en pour affiner un peu notre borne du nombre de clauses satisfiables. Si on détecte deux clauses non satisfaites qui ont pour seul littéral indéterminé respectivement x et $\neg x$, pour une même variable x, ne peuvent pas être satisfaites simultanément. Lorsqu'on rencontre deux telles clauses, on peut donc compter une clause insatisfiable de plus.

Q. 9 Écrire une fonction `clauses\_opposees : cnf -> int -> int` qui prend en argument une formule `phi` simplifiée et l'indice maximal `n` de ses variables propositionnelles, et qui renvoie le nombre de paires de clauses opposées dans `phi` (l'une contenant uniquement un littéral x et l'autre contenant uniquement son littéral opposé $\neg x$) dans phi. On suppose ici que chaque clause distincte n'apparait qu'une seule fois dans `phi`.

Q. 10 En déduire une fonction `insat2 : cnf -> int -> int` qui prend en argument une formule `phi` simplifiée et l'indice maximal `n` de ses variables propositionnelles, et qui renvoie une borne inférieure sur le nombre de clauses insatisfiables de `phi`, en comptant les clauses vides et les clauses opposées.

Q. 11 Écrire une fonction `maxSat2 : cnf -> bool array` qui renvoie une valuation optimale pour la formule donnée en argument, en implémentant un Branch and Bound optimisé. L'exploration devra simplifier `phi` à chaque étage de l'arbre d'exploration, et compter le nombre de clauses insatisfiables par les méthodes précédentes.

B NP-complétude de Max2SAT

On va montrer dans cette partie que MaxSAT est difficile à résoudre efficacement, car il s'agit d'un problème NP-difficile. On peut même montrer plus fort que ça, le problème reste NP-difficile si on le restreint aux formules en CNF dont chaque clause contient au plus k littéraux (MAXkSAT), dès que $k \geqslant 2$. Notez que le problème MAXkSAT est donc "plus difficile à résoudre" que le problème k-SAT, qui est NP-difficile dès que $k \geqslant 3$ mais qu'on peut résoudre en temps linéaire pour $k = 2$. Intéressons nous donc au problème MAx2SAT.

On définit le problème d'optimisation MAX2SAT de la manière suivante :
Instance : une formule propositionnelle φ en forme normale conjonctive telle que chaque clause contient au plus 2 littéraux
Solution admissible : une valuation v (sur les variables de φ)
Mesure à maximiser : nombre de clauses de φ satisfaites par v.

Q. 12 Donner le problème de décision à seuil associé au problème Max2SAT

On va montrer que Max2Sat (à seuil) est NP-difficile en utilisant une réduction polynomiale depuis 3-SAT.

Q. 13 On se donne trois littéraux l_1,l_2 et l_3 et une variable x indépendante des trois littéraux. On note $\overline{l_1}$ la négation de l_1. Ainsi : $\overline{x_1} = \neg x_1$ et $\overline{\neg x_1} = x_1$. On considère alors le groupe de 10 clauses suivant : $l_1 \wedge l_2 \wedge l_3 \wedge x \wedge \left(\overline{l_1} \vee \overline{l_2}\right) \wedge \left(\overline{l_2} \vee \overline{l_3}\right) \wedge \left(\overline{l_1} \vee \overline{l_3}\right) \wedge (l_1 \vee \neg x) \wedge (l_2 \vee \neg x) \wedge (l_3 \vee \neg x)$

(a) Montrer que si $l_1 \vee l_2 \vee l_3$ est valide, alors on peut donner à x une valeur telle que 7 clauses sont satisfaites, mais pas plus. (Autrement dit, si on dispose d'une valuation v sur les variables propositionnelles de l_1,l_2 et l_3 qui satisfait $l_1 \vee l_2 \vee l_3$, alors on peut étendre v sur x de sorte à satisfaire 7 clauses, mais pas plus.)

(b) Montrer que si $l_1 \vee l_2 \vee l_3$ n'est pas valide, alors on ne peut pas satisfaire plus de 6 clauses.

Q. 14 Étant donnée une formule 3SAT φ avec m clauses, définir une formule Max2SAT φ de taille polynomiale et un seuil k tel que φ est satisfiable si et seulement s'il existe une valuation pour φ satisfaisant au moins k clauses.

C Solution approchée

C. 1 Algorithme probabiliste

Puisque trouver une solution optimale au problème MAXSAT en temps polynomial semble difficilement faisable, nous allons maintenant nous intéresser à des moyens efficaces de trouver des solutions approchées. Regardons ce qui se passe lorsqu'on prend une valuation au hasard.

Soit φ une formule propositionnelle en forme normale conjonctive avec m clauses et n variables. Soit v une valuation telle que les valeurs de vérité des n variables propositionnelles de φ sont données par n tirages aléatoires indépendants, chaque variable étant associée à V avec une probabilité $\frac{1}{2}$. On note X_i la variable aléatoire qui vaut 1 si $v(x_i) = \mathrm{V}$ et 0 sinon.

On se demande alors à quel point la valuation obtenue est une bonne solution du problème MAXSAT, c'est à dire combien de clauses elle satisfait. On note $sat(v,\varphi)$ le nombre de clauses satisfaites par v dans φ. $sat(v,\varphi)$ est donc une variable aléatoire.

Q. 15 Montrer que si chaque clause de φ contient au moins k littéraux indépendants (c'est-à-dire qui portent tous sur des variables propositionnelles différentes), alors l'espérance du nombre de clauses satisfaites par la valuation aléatoire v vérifie $\mathbb{E}(\mathrm{sat}(v,\varphi)) \geqslant m\left(1 - \frac{1}{2^k}\right)$.

Q. 16 En supposant que φ ne contient aucune clause vide (quitte à supprimer les clauses vides de φ), donner une minoration de l'espérance du nombre de clauses satisfaites.

On considère alors l'algorithme suivant :

Pseudo-code

```
MaxSat_proba(φ):
  Pour chaque x_i variable aléatoire de φ :
    Tirer aléatoirement (et uniformément) une valeur de vérité X_i pour x_i
  v ← valuation qui à chaque x_i associe X_i
  Renvoyer v
```

Q. 17 L'algorithme ci-dessus est-il une $\frac{1}{2}$-approximation pour le problème Max2SAT? Et une $\frac{1}{4}$-approximation? Si oui, le prouver. Si non, donner des contre-exemples.

Q. 18 Écrire une fonction `maxSat_proba : cnf -> bool array` qui implémente l'algorithme `MAXSAT_PROBA` et renvoie une valuation aléatoire.

C. 2 Algorithme d'approximation pour MaxSAT

Dans cette partie, nous allons voir un moyen de dérandomiser l'algorithme précédent, c'est-à-dire supprimer le caractère aléatoire, de manière à garantir que le nombre de clauses satisfaites est supérieur ou égal à l'espérance du tirage aléatoire.

Pour chaque variable x_i, on va lui attribuer la valeur V ou F selon ce qui maximise l'espérance du nombre de clauses satisfaites en sachant les valeurs de vérité qu'on a déjà donné aux variables précédentes x_j avec $j < i$.

On a donc besoin de la notion d'espérance conditionnelle, définie comme suit :
Soient X une variable aléatoire réelle sur un univers Ω fini et A un événement de probabilité non nulle. On définit l'espérance conditionnelle de X sachant A par :

$$\mathbb{E}(X \mid A) = \sum_{x \in X(\Omega)} x\mathbb{P}(X = x \mid A)$$

Ainsi, en sachant qu'on a déjà construit une valuation partielle v_i sur les variables x_1 à x_i, l'espérance conditionnelle $\mathbb{E}\left(\text{sat}(v,\varphi) \mid v_i\right)$ du nombre de clauses satisfaites sachant v_i vérifie :

$$\mathbb{E}\left(sat(v,\varphi) \mid v_i\right) = \sum_{C \text{ clause de } \varphi} P\left(C \text{ est satisfaite } \mid v_i\right)$$

La probabilité conditionnelle $P\left(C \text{ est satisfaite } \mid v_i\right)$ qu'une clause C soit satisfaite dépend des littéraux de C et des valeurs de vérité déjà affectées à certains de ces littéraux :

- Si l'un des littéraux est défini par v_i et vaut V , alors $P\left(C \text{ est satisfaite } \mid v_i\right) = 1$.
- Si tous les littéraux sont définis par v_i et valent F , alors $P\left(C \text{ est satisfaite } \mid v_i\right) = 0$
- S'il y a k littéraux non définis (et aucun défini à V), alors $P\left(C \text{ est satisfaite } \mid v_i\right) = 1 - \frac{1}{2^k}$

On considère alors l'algorithme suivant :

Pseudo-code

```
MaxSat_approx(φ):
  Pour chaque x_i variable aléatoire de φ de x_1 à x_n :
    E_F ← espérance du nombre de clauses satisfaites par un tirage aléatoire des
      variables x_{i+1} à x_n , connaissant les valeurs déjà fixées pour x_1 à x_{i-1} et
      en donnant à x_i la valeur F
    E_V ← espérance du nombre de clauses satisfaites par un tirage aléatoire des
      variables x_{i+1} à x_n , connaissant les valeurs déjà fixées pour x_1 à x_{i-1} et
      en donnant à x_i la valeur V
    X_i ← V si E_V ⩾ E_F et F sinon.
  v ← valuation qui à chaque x_i associe X_i
  Renvoyer v
```

Q. 19 Montrer que la propriété « $\mathbb{E}\left(\text{sat}(v,\varphi) \mid v_i\right) \geqslant \mathbb{E}(\text{sat}(v,\varphi))$ » est un invariant de la boucle "Pour" de l'algorithme MaxSat_APPROX.

Q. 20 En déduire que l'algorithme `MaxSat_APPROX` calcule une valuation v satisfaisant dans φ un nombre de clauses supérieur ou égal à l'espérance $\mathbb{E}(\text{sat}(v,\varphi))$ du nombre de clauses satisfaites.

Q. 21 Écrire une fonction `maxSat_approx : cnf -> bool array` qui prend en argument une formule propositionnelle `phi` et renvoie une valuation obtenue par l'algorithme précédent.
Attention aux calculs sur les flottants! On conseille de tout multiplier par $2^{k_{max}}$, avec $k_{\max}$ le nombre maximal de littéraux dans une clause de φ, pour ne manipuler que des entiers. On compare alors de manière équivalente $k_{\max}$ .E_F et $k_{\max}$. E_V plutôt que E_V et E_F.