DS4 Corrigé : type CCINP

I 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.

```
let rec var_max l = match l with
   |[] -> 0
   |litteral :: c' -> max (max litteral (-litteral)) (var_max c')

let rec n_var phi = match phi with
   |[] -> 0
   |c :: phi' -> max (var_max c) (n_var phi')
```

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.

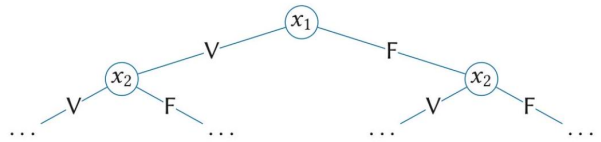
A Solution optimale

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

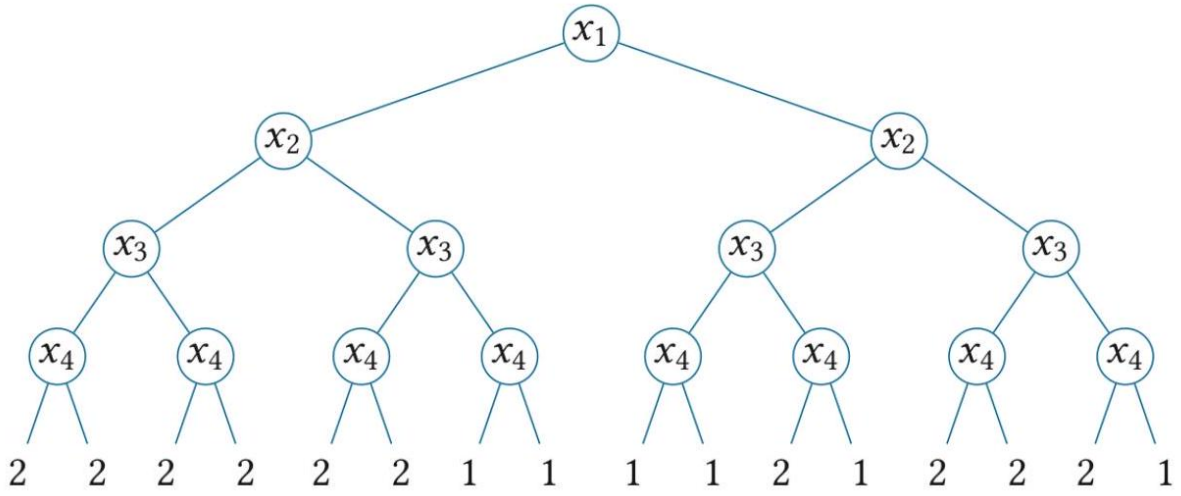
arbre d'exploration de la forme suivante :



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?
Sol : Par exemple, la valuation σ suivante :

$$\sigma(x_1) = F; \sigma(x_2) = V; \sigma(x_3) = \sigma(x_4) = F$$

qui satisfait $m - 1 = 8$ - $1 = 7$ clauses.

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.

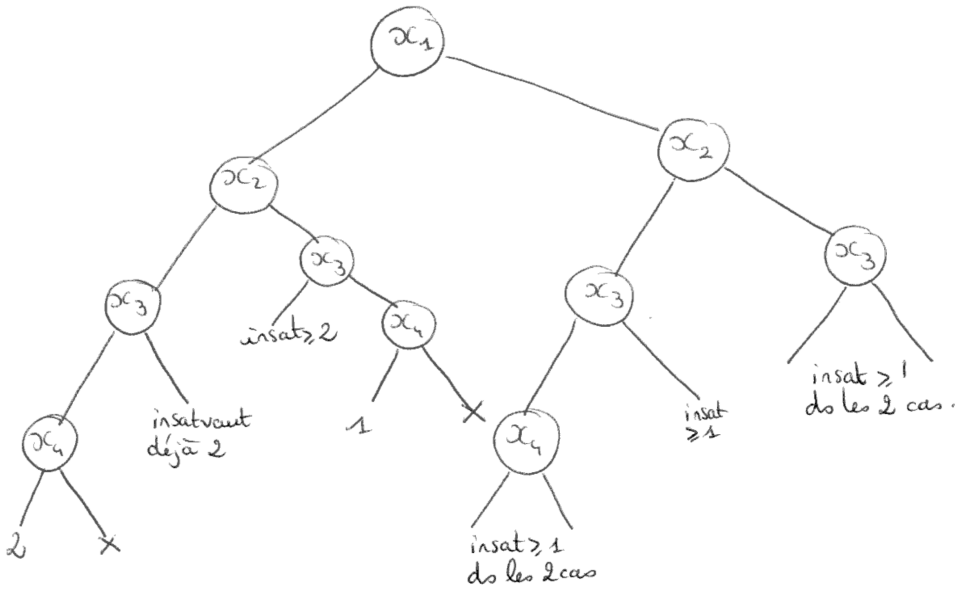
Q. 3 Ecrire une fonction `insat_clause : bool array -> int -> clause -> bool` telle que `insat_clause v k c` prend en argument une valuation partielle `v` portant sur les variables x_1 à x_k, et une clause `c` ; et renvoie `true` si la clause `c` ne peut pas être satisfiable par une valuation étendant `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.

```
let insat_clause v (k : int) (c : clause) =
   let verifie_litteral i =
      if i > 0 then i < k && not v.(i)
      else -i < k && v.(-i)
   in List.for_all verifie_litteral c
```

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`.

```
let rec insat v (k : int) (phi : cnf) = match phi with
   |[] -> 0
   |c :: phi' -> if insat_clause v k c then 1 + insat v k phi'
                 else insat v k phi'
```

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.

```
let maxSat phi =
   let n = n_var phi in
   let v_opt = ref [| |] in (* référence de tableau, on y stockera le meilleur tableau vu *)
   let nb_opt = ref (-1) in
   let v = Array.make (n+1) false in
   let rec explore k =
      let sat_v = List.length phi - insat v k phi in
      if k = n + 1 then begin
         if sat_v > !nb_opt then begin
            v_opt := Array.copy v; (* attention à copier le tableau, sinon on continuera
                                      d'écrire par dessus !*)
            nb_opt := sat_v
         end
      end;
```

```
    if (sat_v > !nb_opt) then begin
       v.(k) <- true;
       explore (k+1);
       v.(k) <- false;
       explore (k+1)
    end
  (*sinon on ne fait rien, ça élague la branche*)
  in explore 1; (* 0 est une case fantôme *)
  v_opt
```

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`.

```
let simplifie x (phi : cnf) : cnf =
   let rec simplify_clause c = match c with
       |[]-> []
       |y :: _ when y = x -> raise Exit
       |y :: c when y = -x -> simplify_clause c
       |y :: c-> y :: simplify_clause c in
   let fold_clause acc c =
      try (simplify_clause c)::acc
      with Exit -> acc
   in
   List.fold_left fold_clause [] 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.

```
let clauses_vides phi =
   List.length (List.filter (fun l -> l = []) phi)
```

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`.

```
let clauses_opposees n phi =
   let uc = Array.make (2*n+1) 0 in
   List.iter (fun c -> match c with
      |x::[]-> uc.(n+x) <- uc.(n+x) + 1
      | _ -> ()) phi;
   let o = ref 0 in
   for x = 1 to n do
      o := !o + min uc.(n-x) uc.(n+x)
   done;
   !o
```

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.

```
let insat2 n phi =
   clauses_vides phi + clauses_opposees n phi
```

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.

```
let maxsat2 phi =
   let n = n_var phi in
   let v_opt = ref [| |] in (* référence de tableau, on y stockera le meilleur tableau vu *)
   let nb_opt = ref (-1) in
   let v = Array.make (n+1) false in
   let rec explore k psi =
      let sat_v = List.length phi - insat2 n psi in
      if k = n + 1 then begin
         if sat_v > !nb_opt then begin
            v_opt := Array.copy v; (* attention à copier le tableau,
                                      sinon on continuera d'écrire par dessus !*)
            nb_opt := sat_v
         end
      end;
      if sat_v > !nb_opt then begin
         v.(k) <- true;
         explore (k+1) (simplifie k psi);
         v.(k) <- false;
         explore (k+1) (simplifie (-k) psi)
      end
      (* sinon on ne fait rien, ça élague la branche*)
   in explore 1 phi; (* 0 est une case fantôme *)
   v_opt
```

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
Sol : Etant donnés une formule du calcul propositionnel sous forme 2SAT et un entier k, existe-t-il une valuation v qui permette de satisfaire au moins k clauses de la formule.

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.)

Sol : On raisonne par cas sur le nombre de littéraux valides parmi l_1,l_2 et l_3.
- Si les trois sont valides, alors les clauses l_1,l_2,l_3,$l_1 \vee \neg x$,$l_2 \vee \neg x$ et $l_3 \vee \neg x$ sont valides et les clauses $\neg l_1 \vee \neg l_2$,$\neg l_2 \vee \neg l_3$ et $\neg l_3 \vee \neg l_1$ sont invalides, indépendamment de la valuation de x. Choisir V pour valeur de x valide une septième clause, et choisir F ne permet rien de plus.
- Si deux sont valides. Par symétrie, on suppose qu'il s'agit de l_1 et l_2. Alors les six clauses l_1,l_2,$\neg l_2 \vee \neg l_3$,$\neg l_3 \vee \neg l_1$,$l_1 \vee \neg x$,$l_2 \vee \neg x$ sont valides indépendamment de de la valeur de x. Selon le choix d'une valeur pour x, on valide en plus soit x, soit $l_3 \vee \neg x$, c'est-à-dire 7 au total dans tous les cas.
- Si un seul est valide. Par symétrie, on suppose qu'il s'agit de l_1. Alors les cinq clauses l_1,$\neg l_1 \vee \neg l_2$,$\neg l_2 \vee \neg l_3$,$\neg l_3 \vee \neg l_1$ et $l_1 \vee \neg x$ sont valides indépendamment de la valeur de x. Selon le choix d'une valeur pour x, on valide en plus soit x, soit $l_2 \vee \neg x$ et $l_3 \vee \neg x$, c'est-à-dire 7 au maximum.

(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.
Sol : Si aucune n'est valide, alors les clauses $\neg l_1 \vee \neg l_2$,$\neg l_2 \vee \neg l_3$,$\neg l_1 \vee l_3$ sont valides, et les clauses l_1,l_2 et l_3 ne le sont pas. Selon le choix d'une valeur pour x, on valide en plus soit x, soit $l_1 \vee \neg x$,$l_2 \vee \neg x$ et $l_3 \vee \neg x$, c'est-à-dire 6 au maximum.

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.
Sol : Pour chaque clause $l_1 \vee l_2 \vee l_3$ de notre formule φ, on construit un groupe de dix clauses tel qu'à la question précédente, avec x une variable non encore utilisée (une nouvelle pour chaque clause). Dans le cas d'une clause l_1 unaire ou d'une clause $l_1 \vee l_2$ binaire, on complète d'abord en $l_1 \vee l_1 \vee l_1$ (resp. $l_1 \vee l_1 \vee l_2$) puis on construit le même groupe. On obtient alors une formule φ' comportant $10m$ clauses, et on fixe le seuil $k = 7m$. Si la formule φ est satisfiable, alors il existe une valuation v pour φ, pour laquelle au moins un littéral est valide dans chaque clause. On peut étendre cette valuation pour φ' de sorte que 7 clauses soit valides dans chaque groupe, d'où $7m$ clauses valides au total. À l'inverse, supposons qu'il existe une valuation v' satisfaisant $7m$ clauses de φ'. On a vu que seules 7 clauses au maximum pouvaient être simultanément satisfaites dans chacun des m groupes. La valuation v' satisfait donc exactement 7 clauses par groupe. Par l'analyse de la question précédente, on sait que cela n'est possible que pour une valuation satisfiant la clause de φ correspondante. Donc toutes les clauses de φ sont satisfaites par v' , et la formule est bien satisfiable.

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 $\mathbb{E}(\mathrm{sat}(v,\varphi)) \geqslant m\left(1 - \frac{1}{2^k}\right)$.

Sol : Considérons une clause C avec k littéraux indépendants $l_1,l_2,\ldots,l_k$. Cette clause n'est fausse que si tous ses littéraux le sont, événement dont la probabilité est divisée par deux pour chaque littéral supplémentaire :

$$P(C \text{ est fausse }) = P(\forall i \in \{1,\ldots,k\}, l_i \text{ n'est pas satisfaite })$$

$$P(C \text{ est fausse }) = \prod_{i=1}^{k} P(l_i \text{ n'est pas satisfaite }) = \prod_{i=1}^{k} \frac{1}{2} = \frac{1}{2^k}$$

par indépendance des événements.

La probabilité que cette clause soit satisfaite par notre valuation aléatoire est donc au contraire $1 - \frac{1}{2^k}$, et augmente avec le nombre k de littéraux. Ainsi, si φ ne contient que des clauses avec au moins k littéraux indépendants, l'espérance du nombre de clauses satisfaites par une valuation aléatoire est :

$$E(sat(v,\varphi)) = E(\sum_{j=1}^{m} \mathbb{1}_{\text{ clause } C_j \text{ satisfaite }}) = \sum_{j=1}^{m} P(C_j \text{ est satisfaite }) \geq m(1 - \frac{1}{2^k})$$

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.

Sol : Sans restriction sur le nombre de littéraux par clause, on peut appliquer le théorème précédent avec $k = 1$, et minorer l'espérance par $\frac{m}{2}$. Ainsi, prendre une valuation aléatoire faite de n tirages indépendants donne de bonnes chances de satisfaire un nombre de clauses au moins égal à la moitié du nombre maximal de clauses satisfiables simultanément (et même plus, si la plupart des clauses ne sont pas trop petites).

On considère alors l'algorithme suivant :

Pseudo-code

```
1 MaxSat_proba(φ):
2   Pour chaque x_i variable aléatoire de φ :
3     Tirer aléatoirement (et uniformément) une valeur de vérité X_i pour x_i
4   v ← valuation qui à chaque x_i associe X_i
5   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.
Sol : Non, absolument pas ! Si on n'a pas de chance, l'algorithme peut renvoyer une valuation qui ne satisfait aucune clause alors que toutes sont satisfiables simultanément. Par exemple, tout simplement, la formule $\varphi = x_1 \wedge x_2$ contenant deux clauses à une variable chacune. L'algorithme

peut associer toutes les variables à false, auquel cas aucune clause n'est satisfaite, alors qu'on peut en satisfaire 2 : $v_{renvoye} = 0 < \frac{1}{2} opt = 1$ (idem avec 1/4).

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

```
let maxSat_proba phi =
   let n = n_var phi in
   Array.init (n+1) (fun _ -> Random.int 2)
```

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 données 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

```
1  MaxSat_approx(φ):
2    Pour chaque x_i variable aléatoire de φ de x_1 à x_n :
3      E_F ← espérance du nombre de clauses satisfaites par un tirage aléatoire des
4        variables x_{i+1} à x_n , connaissant les valeurs déjà fixées pour x_1 à x_{i-1} et
5        en donnant à x_i la valeur F
6      E_V ← espérance du nombre de clauses satisfaites par un tirage aléatoire des
7        variables x_{i+1} à x_n , connaissant les valeurs déjà fixées pour x_1 à x_{i-1} et
8        en donnant à x_i la valeur V
9      X_i ← V si E_V ⩾ E_F et F sinon.
10   v ← valuation qui à chaque x_i associe X_i
11   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.
Sol : En notant v_i la valuation partielle construite après i tours de boucle, on démontre que la propriété : $E(sat(v,\varphi)|v_i) \geq E(sat(v,\varphi))$ est un invariant de la boucle Pour de l'algorithme.

- Initialisation : À l'entrée de la boucle, la valuation partielle v_0 est vide et on a $E(sat(v,\varphi)|v_i) = E(sat(v,\varphi))$.

– Conservation : Supposons que $E(sat(v,\varphi)|v_i) \geq E(sat(v,\varphi))$. Au tour $i+1$, l'algorithme complète v_i en une valuation partielle v_{i+1} telle que :
$E(sat(v,\varphi)|v_{i+1}) = \max(E(sat(v,\varphi)|v_i,x_{i+1} = V),E(sat(v,\varphi)|v_i,x_{i+1} = F))$.
Or l'événement $x_{i+1} = V$ a une probabilité $P(x_{i+1} = V) = \frac{1}{2}$ indépendante de v_i, car on regarde un tirage aléatoire v sachant les valeurs fixées jusqu'à x_i incluse. Donc par linéarité de l'espérance, on a: $E(sat(v,\varphi)|v_i) = \frac{1}{2}E(sat(v,\varphi)|v_i,x_{i+1} = V) + \frac{1}{2}E(sat(v,\varphi)|v_i,x_{i+1} = F)$
L'une des deux espérances conditionnelles $E(sat(v,\varphi)|v_i,x_{i+1} = V)$ ou $E(sat(v,\varphi)|v_i,x_{i+} = F)$ est donc supérieure ou égale à $E(sat(v,\varphi)|v_i)$. Comme v_{i+1} est choisie telle que $E(sat(v,\varphi)|v_{i+1})$ soit la plus grande de ces deux valeurs, on peut conclure que :
$E(sat(v,\varphi)|v_{i+1}) \geq E(sat(v,\varphi)|v_i) \geq E(sat(v,\varphi))$.

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.
Sol : La propriété $E(sat(v,\varphi)|v_i)$ étant un invariant, elle est encore vraie après le dernier tour de boucle. À ce moment-là, on a donc $E(sat(v,\varphi)|v_n) \geq E(sat(v,\varphi))$, avec v_n une valuation donnant une valeur de vérité à chacune des n variables de φ. La valuation v_n étant totale, la valeur de chaque clause est définie, et l'espérance $E(sat(v,\varphi)|v_n)$ est donc précisément le nombre de clauses satisfaites par v_n. Ceci conclut la démonstration, puisque l'algorithme renvoie alors la valuation v_n, qui satisfait $E(sat(v,\varphi)|v_n) \geq E(sat(v,\varphi)) \geq \frac{m}{2}$ clauses de φ.

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.

```
let maxSat_approx phi =
   let n = n_var phi in
   let v = Array.make (n + 1) false in
   let k = List.fold_left max 0 (List.map List.length phi) in
   let b = 1 lsl k in (* shift 1 de k bits vers la gauche, i.e. b = 2^k *)
   (* on pourrait le calculer par exponentiation rapide *)
   for i = 1 to n do
   (* probabilité conditionnelle de satisfaction d'une clause *)
      let expect_cl cl =
         let sat_literal x = abs x <= i && x > 0 = v.(abs x) in
         if List.exists sat_literal cl then b
         else let cl' = List.filter (fun x -> abs x > i) cl in
         b - b / (1 lsl List.length cl') (* idem, c'est 2^nb_litteraux*)
      in
     (* espérance conditionnelle *)
     let expect () = List.fold_left (+) 0 (List.map expect_cl phi) in
     let exp_f = expect () in
     v.(i) <- true;
     let exp_t = expect () in
     if exp_f > exp_t then v.(i) <- false
   done;
   v
```