Exercice 1

Après s'être connecté à la session Linux, vous créerez un dossier personnel à votre nom dans le répertoire `Home`. Vous utiliserez ce dossier pour créer vos fichiers.

1. Créer un fichier `hello.c`, en C, qui affiche un message « Hello world ! », puis le compiler et l'exécuter.
2. Créer un fichier `hello.ml`, en OCaml, qui affiche un message « Hello world ! », puis l'interpréter.
3. Compiler et exécuter le fichier `hello.ml`.

Dans une console, on peut utiliser les commandes de navigation de dossier pour se placer dans le dossier personnel (`cd Nom_du_dossier` pour se déplacer dans un dossier, `cd ..` pour revenir au dossier parent).

Pour interpréter un fichier `.ml`, on pourra, toujours dans la console, taper la commande `utop` (ou `ocaml`) pour lancer un interpréteur. La commande d'interprétation est alors `#use "nom_du_fichier.ml";;`.

Pour compiler un fichier `.ml`, la commande est similaire à celle pour C, en remplaçant `gcc` par `ocamlc` (il n'y a pas toutes les options de compilation, mais celle pour renommer l'exécutable est similaire).

On s'intéresse dans la suite de ce TP à la résolution du problème `MAXSAT` :

* **Instance** : une formule booléenne φ en forme normale conjonctive sur $\mathcal{V} = \{x_0, \ldots, x_{n-1}\}$ contenant m clauses disjonctives $C_1, \ldots, C_m$.
* **Solution** : une valuation $\mu \in \{0,1\}^{\mathcal{V}}$.
* **Optimisation** : maximiser le nombre de clauses satisfaites, c'est-à-dire le cardinal de l'ensemble $J = \{j \in [\![1, m]\!] \mid \mu(C_j) = 1\}$.

On représente en OCaml une formule en FNC sur $\mathcal{V} = \{x_0, \ldots, x_{n-1}\}$ par le type suivant :

```ocaml
type litteral = Var of int | Neg of int;;

type clause = litteral list;;

type fnc = clause list;;
```

où un littéral sera représenté par un objet `litteral`, une liste de littéraux sera interprétée comme une clause disjonctive de ces littéraux, et une liste de clauses disjonctives sera interprétée comme une FNC.

Par exemple, la formule $\varphi = (\overline{x_0} \vee x_1 \vee \overline{x_2}) \wedge (x_0 \vee \overline{x_1} \vee x_2) \wedge (x_0 \vee x_1 \vee x_2) \wedge (\overline{x_0} \vee \overline{x_1})$ sera représentée par :

```ocaml
let phi = [[Neg 0; Var 1; Neg 2];
           [Var 0; Neg 1; Var 2];
           [Var 0; Var 1; Var 2];
           [Neg 0; Neg 1]];;
```

Par ailleurs, on représentera une valuation sur $\mathcal{V} = \{x_0, \ldots, x_{n-1}\}$ par un tableau d'entiers de taille n, contenant des 0 et des 1, tel que si $\mu \in \{0,1\}^{\mathcal{V}}$ est représenté par `mu`, alors `mu.(i)` $= \mu(x_i)$ pour tout $i \in [\![0, n-1]\!]$.

Exercice 2

Le fichier `TP9_formules.txt` (trouvable sur le site `https://github.com/nathaniel-carre/MPI-LLG`)contient des formules en FNC qui seront utilisées pour tester les fonctions suivantes. On cherche à lire ce fichier et à convertir les chaînes de caractères en formules. Ce fichier contient une formule par ligne selon le format suivant :

- un littéral est écrit sous la forme d'un entier ou de la négation d'un entier ;
- une clause correspond à plusieurs littéraux séparés par des virgules ;

– une FNC correspond à plusieurs clauses séparées par des point-virgules.

Par exemple, la formule φ décrite précédemment s'écrirait :

```
-0,1,2;0,-1,2;0,1,2;-0,-1
```

On donne les fonctions suivantes :

– `open_in : string -> in_channel` prend en argument une chaîne de caractère correspondant à un nom de fichier (avec le chemin absolu si besoin, par exemple de la forme `"/Chemin/du/dossier/fichier.txt"`), et renvoie un canal de lecture ;
– `input_line : in_channel -> string` prend en argument un canal de lecture et renvoie une chaîne de caractère correspondant à la ligne en cours de lecture, et passe à la ligne suivante sur le canal de lecture, ou lève une exception `End_of_file` si le canal de lecture a atteint la fin du fichier ;
– `close_in : in_channel -> unit` ferme un canal de lecture ;
– `String.split_on_char : char -> string -> string list` prend en argument un caractère `c` et une chaîne de caractères `s` et renvoie une liste de chaînes de caractères $[s_0; \ldots, s_{k-1}]$ correspondant aux sous-chaînes (éventuellement vides) de s séparées par le caractère c, c'est-à-dire telles que $s = s_0 c s_1 c \ldots c s_{k-1}$ et aucun des s_i ne contient le caractère c ;
– `int_of_string : string -> int` convertit une chaîne représentant un entier en cet entier.

1. Écrire une fonction `lire_fichier : string -> string list` qui prend en argument une chaîne de caractères correspondant à un nom de fichier et renvoie la liste des lignes de ce fichier.
2. Écrire une fonction `fnc_ligne : string -> fnc` qui convertit une chaîne de caractère au format décrit précédemment en la FNC correspondante.
3. En déduire une fonction `tab_fnc : string -> fnc array` qui renvoie un tableau des FNC d'un fichier.

Exercice 3

1. Écrire une fonction `taille_V : fnc -> int` qui prend en argument une formule en FNC sur $\mathcal{V}$ et détermine $|\mathcal{V}|$. On supposera sans le vérifier que toutes les variables d'indices entre 0 et $|\mathcal{V}| - 1$ sont utilisées.
2. Écrire une fonction `evaluer : clause -> int array -> int` qui prend en argument une clause disjonctive C et une valuation μ et calcule $\mu(C)$. On supposera sans le vérifier que C et μ sont bien définies sur le même ensemble de variables.
3. En déduire une fonction `taille_J : fnc -> int array -> int` qui prend en argument une formule φ en FNC et une valuation μ et renvoie le nombre de clauses de φ satisfaites par μ.

On peut assimiler une valuation $\mu \in \{0,1\}^{\mathcal{V}}$ à un entier dont on considère la représentation binaire.

4. Écrire une fonction `valuation : int -> int -> int array` telle que `valuation n k` renvoie la valuation sur n variables correspondant à l'entier $k \in [\![0, 2^n - 1]\!]$.
5. En déduire une fonction `maxsat_naif : fnc -> int array` qui résout le problème `MAXSAT`.
6. Tester la fonction sur les trois premières FNC du fichier.
7. Quelle est la complexité d'une telle fonction ?

Exercice 4

On considère l'algorithme qui consiste à choisir aléatoirement et uniformément une valuation et à la renvoyer. Par abus de notation, on note J le cardinal de l'ensemble J.

1. [À faire après le TP] On suppose que chaque clause de φ contient au moins k littéraux indépendants (donc portant sur des variables différentes). Montrer que $\mathbb{E}(J) \geqslant m\left(1 - \frac{1}{2^k}\right)$.

2. [À faire après le TP] En déduire que l'algorithme précédent est un algorithme Monte Carlo qui renvoie une valuation qui satisfait au moins la moitié des clauses avec probabilité au moins $\frac{1}{2}$.

On rappelle que `Random.int k` renvoie un entier choisi aléatoirement et uniformément entre 0 et $k-1$.

3. Écrire une fonction `maxsat_alea : fnc -> int array` qui applique l'algorithme décrit précédemment.
4. Écrire une fonction `simulation : fnc -> int -> float` qui prend en argument une formule et un entier m et fait m simulations du calcul précédent, puis renvoie la moyenne du ratio $\frac{J}{J^*}$, J étant l'ensemble de clauses satisfaites par une valuation renvoyée par l'algorithme précédent et J^* un ensemble de taille maximal.

   Indication : on pourra utiliser `float_of_int` pour convertir en flottant
5. Tester la fonction sur les trois premières FNC du fichier.

Malheureusement, cet algorithme probabiliste n'est pas une 2-approximation de `MAXSAT`, car il ne garantit pas que le cardinal de J vérifie la même inégalité que son espérance. On propose de dérandomiser l'algorithme précédent, quitte à augmenter légèrement sa complexité.

Exercice 5

Pour $i \in [\![0, n-1]\!]$, on note μ_i une valuation partielle, c'est-à-dire définie sur $\{x_0, \ldots, x_i\}$ mais non définie sur $\{x_{i+1}, \ldots, x_{n-1}\}$. On note $\mathbb{E}(J \mid \mu_i)$ le nombre moyen de clauses satisfaites en complétant μ_i de manière aléatoire uniforme pour les variables $\{x_{i+1}, \ldots, x_{n-1}\}$ (c'est une espérance conditionnelle). On considère l'algorithme suivant :

Début algorithme
  $\mu \leftarrow$ valuation vide.
  Pour $i = 0$ à $n-1$ **Faire**
    Choisir $\mu(x_i)$ tel que $\mathbb{E}(J \mid \mu_i)$ est maximal.
  Renvoyer μ.

1. [À faire après le TP] Montrer que pour $i \in [\![0, n-2]\!]$:

$$\mathbb{E}(J \mid \mu_i) = \frac{1}{2}\left(\mathbb{E}(J \mid \mu_i, \mu(x_{i+1}) = 1) + \mathbb{E}(J \mid \mu_i, \mu(x_{i+1}) = 0)\right)$$

2. [À faire après le TP] En déduire que la propriété $\mathbb{E}(J \mid \mu_i) \geqslant \mathbb{E}(J)$ est un invariant de boucle dans l'algorithme précédent.
3. [À faire après le TP] En déduire que l'algorithme décrit précédemment est une 2-approximation de `MAXSAT`.

Pour calculer $\mathbb{E}(J \mid \mu_i)$, on remarque que $\mathbb{E}(J \mid \mu_i) = \sum_{ij=1}^{m} \mathbb{P}(C_j \mid \mu_i)$ où $\mathbb{P}(C_j \mid \mu_i)$ désigne la probabilité que la clause C_j soit satisfaite si on complète μ_i de manière aléatoire uniforme pour les variables $\{x_{i+1}, \ldots, x_{n-1}\}$. On remarque que :

- si C_j contient un littéral sur une variable x_0, … x_i évalué à vrai, alors $\mathbb{P}(C_j \mid \mu_i) = 1$;
- si C_j ne contient que des littéraux sur des variables $x_0, \ldots, x_i$ évalués à faux, alors $\mathbb{P}(C_j \mid \mu_i) = 0$;
- si C_j ne contient aucun littéral évalué à vrai sur des variables x_0, …, x_i et contient k littéraux sur des variables x_{i+1}, …, x_{n-1}, alors $\mathbb{P}(C_j \mid \mu_i) = 1 - \frac{1}{2^k}$.

4. Écrire une fonction `proba_condi : clause -> int array -> int -> float` qui prend en argument une clause, une valuation partielle μ_i et l'entier i et calcule et renvoie $\mathbb{P}(C_j \mid \mu_i)$.
5. En déduire une fonction `esperance_condi : fnc -> int array -> int -> float` qui prend en argument une formule, une valuation partielle μ_i et l'entier i et calcule et renvoie $\mathbb{E}(J \mid \mu_i)$.

6. En déduire une fonction `maxsat_2approx : fnc -> int array` correspondant à la 2-approximation décrite précédemment.
7. Quelle est la complexité de cette fonction ?

Exercice 6

On cherche à mettre en œuvre un algorithme *Branch and Bound* pour résoudre de manière exacte le problème `MAXSAT` plus efficacement qu'avec la méthode naïve. On considère pour cela une exploration de l'arbre des possibilité, correspondant à un arbre binaire de hauteur $|\mathcal{V}|$, chaque nœud x_i ayant pour fils les affectations $\mu(x_i) = 0$ ou $\mu(x_i) = 1$.

Pour l'heuristique d'évaluation, on fait le constat suivant :

- une clause dont tous les littéraux sont déjà affectés à faux ne peut pas être satisfaite ;
- si deux clauses C_j et $C_{j'}$ non satisfaites par $\{x_0, \ldots, x_{i-1}\}$ possèdent chacune un seul littéral indéterminé, l'une x_i et l'autre $\overline{x_i}$, alors elles ne peuvent pas être simultanément satisfaites.

1. Écrire une fonction `heuristique : fnc -> int array -> int -> int` qui prend en argument une formule, une valuation partielle μ_i et l'entier i et renvoie une borne supérieure du nombre de clauses qui peuvent être satisfaites selon l'heuristique décrite précédemment.

Pour l'heuristique de branchement, on propose simplement d'affecter $\mu(x_i)$ à 1 en premier si x_i apparaît plus de fois que $\overline{x_i}$ dans des clauses non satisfaites, et 0 sinon.

2. Écrire une fonction `branchement : fnc -> int array -> int -> int` qui prend en argument une formule, une valuation partielle μ_i et l'entier i et renvoie 1 ou 0 selon que x_i apparaît plus de fois que $\overline{x_i}$ dans des clauses non satisfaites ou non.
3. En déduire une fonction `maxsat_bnb : fnc -> int array` qui détermine une valuation optimale selon un algorithme *Branch and Bound*.
4. Vérifier qu'on peut calculer des solutions pour l'ensemble des FNC du fichier.
5. Calculer les ratios $\frac{J}{J^*}$ obtenus pour l'ensemble des FNC du fichier, en comparant l'algorithme d'approximation et l'algorithme précédent.