DS2 option info MPSI
Algorithme polynomial pour le problème 2-SAT

Ce sujet a pour objectif d'arriver à un algorithme polynomial pour résoudre le problème 2-SAT. Même si c'est l'objet d'étude du sujet, aucune connaissance n'est requise sur le problème SAT ici : on va simplement manipuler des formules logiques et des graphes.

Les questions particulièrement difficiles sont marquées d'une étoile.

A minima, les fonctions auxiliaires devront être annotées d'un commentaire indiquant leur spécification.

I) Formules en 2-CNF

Définition 1 [Formule sous forme 2-FNC]

> Une formule logique est dites en 2-FNC si elle est de la forme :
>
> $$\varphi = \bigwedge_{i=1}^{n} p_i \lor q_i$$
>
> où p_i et q_i sont des littéraux, c'est-à-dire soit des variables x, soit des négations de variables $\neg x$.
>
> Les $p_i \lor q_i$ sont appelés les clauses.

Exemple 2

La formule $\varphi = (a \lor b) \land (\neg a \lor \neg b) \land (c \lor \neg c)$ est en 2-FNC.

Les formules en 2-FNC seront représentées en OCaml par le type suivant :

```
type litteral = Var of int | Not of int;;
type formule = (litteral * litteral) list;;
```

On va considérer que les variables d'une formule sont numérotées par les entiers $0, 1, \dots, n-1$, où n est le nombre de variables distinctes de la formule.

Par exemple, en numérotant a, b et c par 0, 1 et 2, la formule φ est représentée par la liste :

```
let phi: formule = [(Var 0, Var 1); (Not 0, Not 1); (Var 2, Not 2)];;
```

On va s'intéresser à la résolution de formules en 2-FNC, c'est-à-dire de trouver une valuation telle que la formule soit vrai, si c'est possible.

On dit qu'une formule est *satisfiable* si une telle valuation existe.

Q1) La formule φ de l'exemple 2 est-elle satisfiable ? Si oui, donnez une valuation des variables telle qu'elle s'évalue à Vrai.

Oui, par exemple avec a = vrai, b = faux, c = vrai.

Q2) Toutes les formules peuvent-elles être mises sous la forme 2-FNC ? Vous pouvez par exemple étudier la formule $a \lor b \lor c$ pour répondre.

Non, par exemple $a \lor b \lor c$ ne peut pas être mise sous cette forme. En effet, si c'était le cas et qu'on avait une telle formule $f \equiv a \lor b \lor c$, la première clause qui n'est pas toujours vraie de f, $p \lor q$, qui contient deux variables, qu'on peut supposer par symétrie être a et b. Alors, si on prend des valeurs

de a et b telles que $p \vee q$ est fausse, et c qui est vrai, on a f qui est fausse alors que $a \vee b \vee c$ est vrai. Absurde !

Q3) Écrire une fonction `neg : litteral -> litteral` qui renvoie la négation d'un littéral (c'est-à-dire qu'elle transforme a en $\neg a$ et $\neg a$ en a).

```
let neg l = match l with
    |Neg i -> Var i
    |Var i -> Neg i;;
```

Q4) Écrire une fonction `nb_litteraux : formule -> int` qui calcule le nombre de littéraux dans une `formule` (en 2-FNC toujours, comme sur tout le reste du sujet).

```
let nb_litteraux f = 2 * List.length f;;
```

Q5) Écrire une fonction `nb_variables : formule -> int` qui calcule le nombre de variables différentes dans une `formule`.

*Ne cherchez pas trop loin ! En particulier, les variables sont des entiers **consécutifs**...*

```
let extract (l: litteral): int = match l with
    |Neg i | Var i -> i;;
let rec nb_variables f = match f with
    |[] -> 0
    |(l1, l2)::q -> max (extract l1) (max (extract l2) (nb_variables q));;
```

Pour représenter une valuation, on va utiliser un tableau. Comme on a déjà numéroté les variables entre 0 et $n-1$, on peut utiliser cette même numérotation pour la valuation, et placer à l'emplacement i du tableau la valeur de la variable i.

```
type valuation = bool array;;
```

Ainsi, pour la formule φ, la valuation :

a	Vrai
b	Vrai
c	Faux

s'écrit en OCaml :

```
let val: valuation = [|true; true; false|]
```

Q6) Écrire une fonction `eval: formule -> valuation -> bool` qui évalue une formule avec une valuation.

On pourra supposer que la valuation contient bien toutes les variables de la formule.

```
let rec eval_litt l v = match l with
    |Var i -> v.(i)
    |Neg i -> not v.(i);;

let rec eval f v = match f with
    |[] -> true
    |(l1, l2)::q -> ((eval_litt l1 v) || (eval_litt l2 v)) && eval q v
```

II) Graphe d'implication

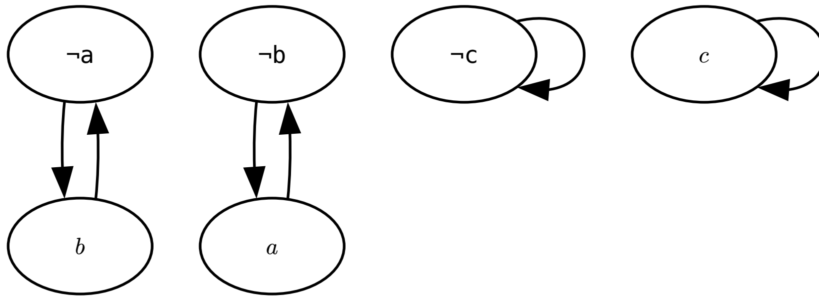
Pour résoudre de telles formules, on va utiliser une construction appelée *graphe d'implication.*

II.1) Définition

Définition 3 [Graphe d'implication]

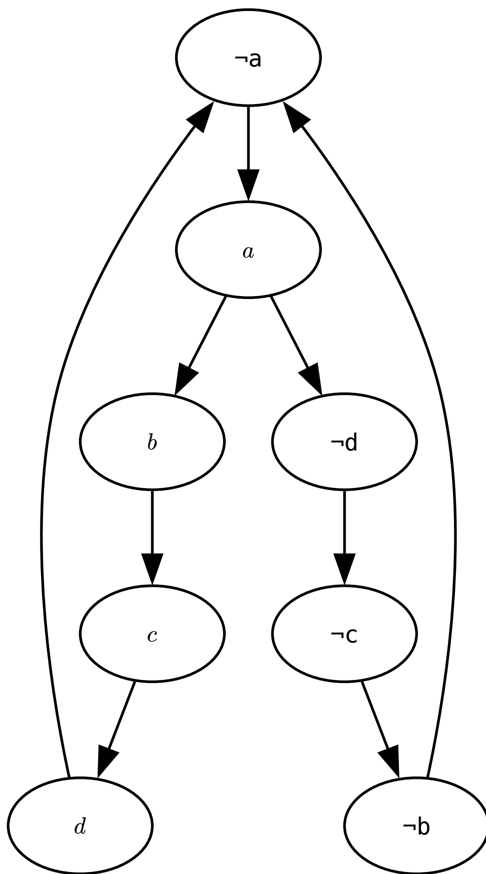
> Pour une formule φ en 2-FNC, on appelle **graphe d'implication** le graphe orienté $G = (S, A)$ dont :
> - les sommet S sont les littéraux de φ (c'est-à-dire, toutes les variables de φ et leurs négations)
> - pour chaque clause $p \lor q$, on place les arrêtes $\neg p \to q$ et $\neg q \to p$
>
> *NB : en fait, on peut voir la clause $p \lor q$ comme une des implications $\neg q \to p$ ou $\neg q \to p$, et donc les arrêtes de graphe sont des implications.*

Par exemple, le graphe correspondant à la formule φ est :



Q7) Représentez le graphe d'implication correspondant à la formule

$$\psi = (a \vee a) \wedge (\neg a \vee b) \wedge (\neg b \vee c) \wedge (\neg c \vee d) \wedge (\neg d \vee \neg a)$$



Q8) Que peut-on en déduire sur la formule si, pour une variable x, le graphe contient une arrête $x \to \neg x$ et une arrête $\neg x \to x$? La formule est-elle alors satisfiable ?

Alors, c'est que la formule contient les clauses $x \vee x$ et $\neg x \vee \neg x$, et elle c'est pas satisfiable. Alternativement, c'est que la formule implique $x \leftrightarrow \neg x$.

Q9) Montrer que, pour x une variable, si le graphe contient un chemin entre x et $\neg x$ et un chemin entre $\neg x$ et x, alors la formule n'est pas satisfiable.

On a un chemin $x \to l_1 \to \ldots \to l_i \to \neg x \to l_{i+1} \to \ldots \to l_k \to x$.

On a donc des clauses $x \to l_1$, $l_1 \to l_2$, et ainsi de suite jusqu'à $l_k \to x$, qui sont toutes en conjonction entre elles et avec le reste de la formule.

Par associativité de l'implication / par récurrence / par un calcul, on a : $x \to \neg x$ et $\neg x \to x$, et donc la formule n'est pas satisfiable.

On admet dans un premier temps que la réciproque est également vraie.

II.2) Implémentation en OCaml

On choisi de représenter les graphes par liste d'adjacence.

Les sommets correspondant aux variables sont numérotés de 0 à $n-1$, et ceux correspondant à leurs négations de n à $2n-1$.

On peut alors utiliser le type suivant pour représenter les graphes :

```
type graphe = int list array
```

où l'élément d'indice i du tableau contiens la listes des voisins de i, c'est-à-dire les sommets v tels que $i \to v$.

Q10) Écrire une fonction `index: litteral -> int` qui renvoie l'indice du sommet correspondant à un littéral dans le graphe.

```
let index l n = match l with
    |Var i -> i
    |Neg i -> n + i
```

Petite erreur de signature dans cette fonction, il aurait fallu prendre n en argument. Toutes les variantes qui permettaient d'avoir accès à n étaient acceptées.

Q11) Écrire une fonction `f2g : formule -> graphe` qui calcule et renvoie le graphe d'implication d'une formule.

```
let f2g f =
    let n = nb_variables f in
    let g = Array.make (2*n) [] in
    let rec ajoute f = match f with
        |[] -> ()
        |(l1, l2)::q -> (
            let i11 = index (neg l1) n in
            let i12 = index (neg l2) n in
            let i21 = index l2 n in
            let i22 = index l1 n in
            g.(i11) <- i12::g.(i11);
            g.(i21) <- i22::g.(i21);
            ajoute q)
    in
    ajoute f;
    g
;;
```

Définition 4 [Composantes fortement connexes]

> Pour $G = (S, A)$ un graphe orienté, on appelle *composante fortement connexe* de G tout sous-ensemble *maximal* des sommets $C \in S$ tel que, pour tout $p, q \in C$, il existe un chemin de p à q et de q à p.
>
> Le mot *maximal* veut ici dire qu'il n'y a aucun autre sommet du graphe que l'on peut ajouter à la composante fortement connexe, de manière à ce qu'elle reste fortement connexe.

Q12) Montrer que les composantes fortement connexes d'un graphe forment une partition de ses sommets, c'est-à-dire que :
1. elles sont disjointes
2. tout sommet appartient à une composante connexe

1. Soit C_1 et C_2 deux composantes connexes qui ont un sommet en commun x. Soit $y \in C_1$ et $z \in C_2$. Alors il existe des chemins de x à y et de y à x, et de même pour z. Donc il y a un chemin $y \to x \to z$ et $z \to x \to y$. Donc, comme les composantes connexes sont maximales, $y \in C_2$ et $x \in C_1$. On en déduit que $C_1 = C_2$.
2. Soit x un sommet. Alors, si on considère l'ensemble $\{x\}$, il existe bien un chemin entre chaque paire de sommet de cet ensemble. Donc, dans le pire cas, $\{x\}$ est la composante connexe qui contient x.

Q13) Montrez que la condition énoncée à la question 9 est équivalente à "pour toute variable x, x et $\neg x$ ne sont pas dans la même composante fortement connexe".

Si x et $\neg x$ ne sont pas dans la même composante fortement connexe, alors il existe soit aucun chemin entre x et $\neg x$, soit aucun entre $\neg x$ et x.

Inversement, si x et $\neg x$ sont dans la même composante fortement connexe, ces chemins existent.

(la question était posée dans le mauvais sens, c'est une erreur)

On va donc calculer les composantes fortement connexes de notre graphe.

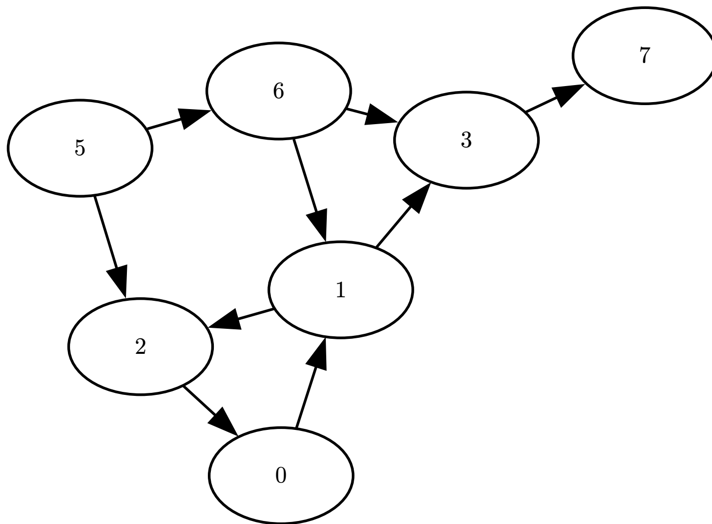
On donne la fonction suivante :

```
let rec parcours (g: graphe) (s: int) (visite: bool array): int list =
    let rec parcours_voisins (l: int list): int list = match l with
        |[] -> []
        |v::q -> (parcours g v visite) @ (parcours_voisins q)
    in
    if visite.(s) then [] else
        (visite.(s) <- true;
        s::(parcours_voisins g.(s)))
;;
```

Q14) Que-fait cette fonction ? Donnez son résultat sur le graphe suivant, en partant de 0, avec le tableau `visite` rempli de `false` au début.

Cette fonction parcours le graphe de sommet en sommet, et renvoie la liste des sommets parcourus dans l'ordre inverse de remontée.

Dans l'exemple, elle renvoie `[0; 1; 2; 3; 7]`



Q15) * Cette fonction ne parcours pas le graphe dans son intégralité. Écrire une fonction `parcours_total: graphe -> int list` qui parcours l'intégralité du graphe avec la fonction parcours (en l'appelant plusieurs fois).

La fonction `parcours_total` doit renvoyer la concaténation des résultats de tous les appels à `parcours`.

On va utiliser un algorithme classique (enfin, "classique", c'est relatif) pour calculer les composantes fortement connexes de notre graphe :

Algorithme 5 [de Kosaraju]

> - Appeler la fonction `parcours_total` et noter l'ordre renvoyé ω, c'est-à-dire l'ordre de remontée du parcours.
> - Transposer le graphe G en G^T (c'est-à-dire inverser toutes ses arrêtes).
> - Parcourir G^T, en suivant l'ordre indiqué par ω, c'est-à-dire que, lorsqu'un appel à `parcours` est terminé, on reprend depuis le premier sommet de ω non visité.
>
> Chaque appel à `parcours` renvoie alors une composante connexe.

Q16) Écrire une fonction `transpose: graphe -> graphe` qui renvoie la transposée d'un graphe, c'est-à-dire que toutes ses arrêtes sont inversées.

Q17) * Écrire une fonction `kosaraju: graphe -> int list list` qui prend en argument un graphe et renvoie la liste de ses composantes connexes (donc une liste de listes de sommets).

Q18) * En combinant les différentes fonctions du sujet (que vous n'avez pas besoin d'avoir traité), écrivez une fonction `satisfiable: formule -> bool` qui détermine si une formule sous forme 2-FNC est satisfiable. Quelle-est sa complexité ?

Q19) Comment peut-on attribuer des valeurs aux variables qui satisfont la formule à l'aide du graphe d'implication ?