

Rapport sur TP noté (résolution de 2-SAT)

Wassim Saidane, Aurélien Authier

Table des matières

1	Nombre maximum de clauses (Exercice 1)	1
2	Formule satisfaisable	1
2.1	Exemples	1
2.1.1	Formule satisfaisable	1
2.1.2	Formule non satisfaisable (Exercice 2)	2
2.2	Algorithme (Exercice 3)	2
3	Graphe orienté associé à une formule	4
3.1	Théorie	4
3.2	Application	5
3.3	Algorithme (Exercice 4)	6
4	Validité d'une formule	7
4.1	Théorie	7
4.2	Algorithme (Exercice 5)	7

Note

Ne connaissant pas les algorithmes des fonctions sage nous avons considéré qu'elles sont constituées d'une seule opération (sauf mention contraire).

1 Nombre maximum de clauses (Exercice 1)

Soit n le nombre de variables et p le nombre de variables dans une clause (ici 2).

Prenons $n = 2$,

On a au maximum 4 variables en prenant en compte les complémentaires, pour trouver le maximum on utilise la combinatoire on a donc 2 parmi 4.

On a donc :

$$\frac{4!}{2! \times 2!} = \frac{24}{4} = 6$$

Prenons $n = 3$

On a au maximum 6 variables et de la même façon on obtient :

$$\frac{6!}{2! \times 4!} = \frac{720}{2 \times 24} = \frac{720}{48} = 15$$

La formule pour déterminer le maximum de clause est donc :

$$\binom{2n}{p} = \left(\frac{2n!}{p! \times (2n - p)!} \right)$$

avec $p = 2$

2 Formule satisfaisable

2.1 Exemples

2.1.1 Formule satisfaisable

Une formule est dite satisfaisable si la formule est évaluée à Vrai.
Exemple (exemple du sujet) :

$$F = (x_1 \vee \neg x_2) \wedge (x_3 \vee x_4) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_4 \vee \neg x_5) \wedge (x_2 \vee \neg x_5)$$

Avec :

$$x_1 = \text{True}$$

$$x_2 = \text{False}$$

$$x_3 = \text{True}$$

$$x_4 = \text{None}$$

$$x_5 = \text{False}$$

2.1.2 Formule non satisfaisable (Exercice 2)

Inversement une formule non satisfaisable est une formule évaluée à Faux
Exemple :

$$F = (x_4 \vee \neg x_3) \wedge (x_4 \vee x_1) \wedge (\neg x_2 \vee \neg x_1) \wedge (x_2 \vee \neg x_5) \wedge (\neg x_2 \vee x_5)$$

Avec : $x_1 = \text{True}$

$x_2 = \text{False}$

$x_3 = \text{True}$

$x_4 = \text{False}$

$x_5 = \text{False}$

2.2 Algorithme (Exercice 3)

Pour implémenter l'algorithme de satisfaisabilité nous avons décidé de représenter l'algorithme avec une sémantique particulière.

Tous d'abord la formule est entrée dans une chaîne de caractère.

Les clauses sont séparées par le symbole "&", et les variables à l'intérieur de la clause sont séparées par le symbole "|".

Lorsque la variable est précédée par "-" cela veut dire qu'il s'agit de la négation.

1 : x_1 -> -1 : $\neg x_1$

Ainsi notre code fonctionne pour les formules 2-SAT. Par exemple en prenant la formule 2-SAT du sujet :

$$F = (x_1 \vee \neg x_2) \wedge (x_3 \vee x_4) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_4 \vee \neg x_5) \wedge (x_2 \vee \neg x_5)$$

On obtient

```
1 F="1|-2&3|4&-2|-3&4|-5&2|-5"
2
```

Les valeurs d'une variable sont affectées à l'aide d'un dictionnaire.

Chaque variable va pointer sur un booléen comme ceci :¹

```
1 dic={}
2 dic[1] = True
3 dic[2] = False
4 dic[3] = True
5 dic[4] = None
6 dic[5] = False
7 dic[6] = False
8 dic[-1] = not(dic[1])
9 dic[-2] = not(dic[2])
10 dic[-3] = not(dic[3])
11 dic[-4] = not(dic[4])
12 dic[-5] = not(dic[5])
13 dic[-6] = not(dic[6])
14
```

1. Nous aurions également pu implémenter une fonction pour attribuer les négations mais ce n'était pas une de nos priorités.

La chaîne de caractère va ensuite être parsée.

Voici une ébauche de notre code permettant d'évaluer une formule

```
1
2     def clauses_liste(F) #O(n)
3
4     def variables_par_clause(l1,l2) #O(n^2)
5
6     def eval_clauses(d1,d2,l) #O(n)
7
8     def eval_2_sat(f,d1):
9         print(f"Affectation : {dic}")
10        clauses = clauses_liste(f) #O(n)
11        print(f"Clauses : {clauses}")
12        result = True
13        clause=[]
14        variables_par_clause(clause,clauses) #o(n^2)
15        print(f"Variables par clause : {clause}")
16        d2={}
17        eval_clauses(d1,d2,clause) #O(n)
18        print(f"Evaluations par clause : {d2}")
19        for value in d2: #O(n)
20            result=(result and d2[value])
21        return result
22
23
```

Complexité : $O(n^2)$

La fonction eval_2_sat prend en paramètre une formule f et un dictionnaire associé. Elle renvoie l'évaluation de la formule (en plus de quelques affichages pour faciliter la lecture).

Tous d'abord nous allons appeler la fonction clauses_liste qui prend une formule 2-sat et va renvoyer une liste (tableau) contenant les clauses.

L'implémentation se fait en une ligne, en effet nous utilisons seulement la fonction split comme ceci :

```
1     def clauses_liste(F):
2         return F.split("&")
3
```

Complexité : $O(n)^2$

Nous splitons donc le tableau par le caractère "&".

Cette liste de clauses va être stockée dans une variable portant le même nom.

Ensuite nous appelons la procédure variables par_clauses ($O(n^2)$ ³) qui va ajouter dans une autre liste la liste des variables non couplées (même procédé que pour la fonction précédente).

Cette liste est appelé clause.

Ensuite nous allons créer un autre dictionnaire qui va affectée une évaluation

2. Il faut parcourir la chaîne de caractères

3. On parcourt une liste et on split (parcours de chaîne de caractères) à chaque élément

pour chaque clause à l'aide de la fonction `eval_clauses`.

```
1     def eval_clauses(d1,d2,l):
2         for i in range(len(l)):
3             v1 = int(l[i][0])
4             v2 = int(l[i][1])
5             d2[tuple(l[i])] = (d1[v1] or d1[v2])
6
```

Complexité : $O(n)^4$

La fonction prend un argument une liste et son dictionnaire et un dictionnaire (vide en théorie).

Nous allons ensuite pour chaque clauses effectuer l'opération logique "ou" entre les deux variables.

On affecte ensuite le couple à un booléen (la valuation).

Enfin nous allons parcourir le dictionnaire qui affecte un booléen par couple (ici `d2`) pour effectuer l'opération logique "et" entre les différentes évaluations des couples.

Cette opération nous donne l'évaluation de la formule complète de la formule stockée dans `result`.

3 Graphe orienté associé à une formule

Une formule 2-SAT peut être transformé en graphe. Ce graphe est appelé **graphe d'implication**.

Un graphe d'implication est un graphe orienté asymétrique. Chaque sommet représente l'état de la variable booléenne, les arrêtes les implications entre ces variables.

3.1 Théorie

Un graphe d'implication à $2n$ sommets (n étant le nombre de variables) et $2m$ arrêtes (m étant le nombre de variables).

Chaque clauses est un "ou" logique mais peut être représentées par deux implications.

Exemple :

$(x_1 \vee x_2)$ peut être représenter par $(\neg x_1 \implies x_2)$ ou $(\neg x_2 \implies x_1)$. On peut donc rajouter les arrêtes $(\neg x_1, x_2)$ et $(\neg x_2, x_1)$

4. Il faut parcourir la liste

3.2 Application

Nous allons prendre comme formule celle du sujet.

$$F = (x_1 \vee \neg x_2) \wedge (x_3 \vee x_4) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_4 \vee \neg x_5) \wedge (x_2 \vee \neg x_5)$$

Nous pouvons déjà déterminer le nombre de sommets et d'arrêtes.

Soit n le nombre de variables, m le nombre de clauses, N le nombre de sommets et M le nombre d'arrêtes.

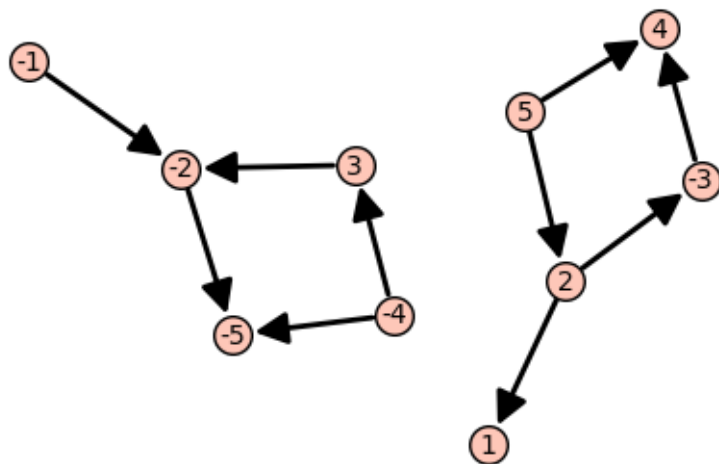
$$n = 5$$

$$m = 5$$

$$N = 2n = 10$$

$$M = 2m = 10$$

Voici le graphe généré par notre code lorsqu'on entre notre formule sur sage (La sémantique décrite précédemment est la même).



Les variables x_1, x_2, x_3, x_4 et x_5 ainsi que leurs négations sont représentées en sommets. Prenons maintenant la première clause.

$(x_1 \vee \neg x_2)$: on obtient donc les implications $\neg x_1 \implies \neg x_2$ et $x_2 \implies x_1$

Le processus est le même pour les autres clauses.

3.3 Algorithme (Exercice 4)

Voici une ébauche de la fonction `formula_2_graph` qui prend en argument un graphe et qui retourne son graphe associé.

Le code

```
1     [...] #O(n^2)
2     g = DiGraph()
3     for var in clause: #O(n)
4         g.add_vertices([int(var[0]), int(var[1])])
5     sommets = g.vertices()
6     for sommet in sommets:
7         g.add_vertices([sommet*-1]) #O(n)
8     [...]
9     for i in range(0, len(clause)): #O(n)
10        clause[i] = [int(clause[i][0]), int(clause[i][1])] #O(n)
11
12    for var in clause: #O(n)
13        g.add_edges([(var[0]*-1, var[1])])
14        g.add_edges([(var[1]*-1, var[0])])
15    return g
```

Complexité : $O(n^2)$ ⁵

Explication

Le tableau `clause` contenant la liste des variables par clauses est obtenu de la même façon que l'exercice précédent. Nous allons ajouter chaque sommets au graphes. Tous d'abord nous allons prendre toutes les variables par clause. A chaque parcours de clause les deux variables sont rajoutées au graphes.

A ce moment là, le graphe n'est pas complet nous allons donc rajouter toutes les négations manquantes. On va donc stocker tous les sommets dans un tableau `sommets`.

En implémentant les formules avec l'utilisation de "-" pour les négations ils nous aient facile de manipuler cette dernière.

En effet, on va parcourir chaque sommet et rajouter sa négation en multipliant par -1⁶. Les variables étant castées en entier précédemment.

Ensuite, nous allons rajouter les arrêtes.

Pour cela nous allons nous intéresser aux clauses contenant un couple de variables. Ces dernières étant stockées en chaine de caractères nous les avons castées en entier pour pouvoir les manipuler.

Après cela nous pouvons effectuer les deux opérations pour rajouter les arrêtes. On parcourt les différentes variables de chaque clause. On rajoute une arrête

5. La fonction `variables_par_classe` est appelé

6. Il existait des variables qui avaient déjà leurs négations représentées sur le graphe. Cela n'est pas dérangeant, les variables ont été écrasées par elle-même.

allant de la négation de la première variable jusqu'à la seconde variable.
 Ensuite on ajoute l'autre arrête entre la négation de la seconde variable et la première variable.
 On va ainsi rajouter 2 arrêtes par clauses.

4 Validité d'une formule

4.1 Théorie

Pour vérifier de la validité d'une formule, nous allons analyser son graphe générer précédemment. En appliquant l'algorithme de Tarjan on obtient un ordre topologique sur les composantes fortement connexe. Soit $CC = [x_1, x_2, \dots, x_n]$ la liste des composantes connexes par tri topologique.

Nous avons donc $\neg(CC) = [\neg x_1, \neg x_2, \dots, \neg x_n]$ qui est également une liste de composantes connexes. La formule est valide si la négation d'une variable n'est pas déjà présente. En considérant les composantes dans l'ordre inverse, on va leur affecter des valuations : tant qu'il reste une composante non évaluée CC , on affecte à tous ses noeuds la valeur True et à tous les noeuds de $\neg CC$ la valeur False.

4.2 Algorithme (Exercice 5)

Voici le code de la fonction `est_valide` :

Le code

```

1  def est_valide(g):
2      temp=list(strongly_connected_components_digraph(g))
3      # Tarjan O(n)
4      liste_connex = temp
5      for connex in liste_connex: #O(n)
6          tab = []
7          for variable in connex: #O(n)
8              variable_temp = variable*-1
9              if tab.__contains__(variable_temp): #O(n)
10                 return False
11             tab.append(variable) #O(1)
12     return True
13

```

Complexité : $O(n^3)$ ⁷

7. La fonction sage : `strongly_connected_components_digraph()` applique l'algorithme de Tarjan qui se fait en temps linéaire

Explication⁸

Nous avons casté l'objet sage de type digraphe⁹ en liste pour pouvoir la manipuler. Ensuite nous allons pour chaque composantes connexes mettre dans un tableau une variable et vérifié si sa négation n'est pas déjà stockée, si elle l'est on retourne False si non on peut réinitialiser le tableau et passer à la prochaine composante fortement connexe.

8. Nous avons pas réussie à implémenter l'affectation
9. Composantes fortement connexes