TP Noté

Résolution de 2-SAT

Auteur :
Nathanaël Bayle
Camille Hounsa

27 Novembre 2020

Table des matières

Chapitre 1

Présentation du projet

1.1 Introduction

Une formule logique 2−SAT est une conjonction de disjonction où chaque clause comporte exactement 2 littéraux. Le but de ce projet est de comprendre et implémenter un algorithme [1, 2] qui permet de décider si une formule est valide.

Le problème est défini sur un ensemble de variable $X = \{x_1, x_2, \ldots, x_n\}$ Un littéral est une variable dans sa forme positive x_i ou sa forme négative $\bar{x_i}$

Une formule 2−SAT est un ensemble de clauses $C = \{C_1, C_2, \ldots, C_m\}$ connectées par des "et" logiques. Le "et" logique est noté $\wedge$.

Chaque clause comporte exactement 2 littéraux connectés par un "ou" logique (noté $\vee$).

Par exemple la formule F suivante :

$$F = (x_1 \vee \bar{x_2}) \wedge (x_3 \vee x_4) \wedge (\bar{x_2} \vee \bar{x_3}) \wedge (x_4 \vee \bar{x_5}) \wedge (x_2 \vee \bar{x_5})$$

Le problème consiste à déterminer s'il existe une assignation pour chaque variable à vrai ou faux de telle manière que l'évaluation de la formule soit vraie.

1.2 SageMath

Le logiciel **SageMath**, appelé communément **Sage**, est un logiciel de calcul mathématique qui utilise **Python** On peut s'en servir pour faire des calculs numériques approchés ou pour faire des calculs exacts d'expressions mathématiques formelles.

SageMath permet de faire des mathématiques générales et avancées, pures et appliquées. Il couvre une vaste gamme de mathématiques, dont l'algèbre, l'analyse, la théorie des nombres, la cryptographie, l'analyse numérique, l'algèbre commutative, la théorie des groupes, la combinatoire, la théorie des graphes, l'algèbre linéaire formelle, etc ...

Nous utiliserons **SageMath** et **Python** tout au long de notre projet afin de répondre au mieux au travail demandé.

1.3 Objectifs

L'objectif principale de ce travail est d'appliquer un algorithme utilisant les propriétés des graphes orientés pour déterminer si une formule 2-SAT est satisfaisable.

On suppose une formule F, de forme normale conjonctive avec au plus deux littéraux par clause. Nous pouvons supposer sans perte de généralité qu'il n'y a aucune clause avec un seul littéral puisque la clause (u) est équivalente à la clause $(u \vee u)$. Pour construire un graphe orienté, nous devons réécrire le problème dans une forme différente pour obtenir un graphe d'implication. On a $u \vee v$ équivalant à $(\bar{u} \Rightarrow v) \wedge (\bar{v} \Rightarrow u)$. Si l'une des deux variables est Fausse, alors l'autre est Vraie.

Pour construire notre graphe orienté d'implication on trace, pour chaque variable x, deux arêtes correspondantes aux implications.

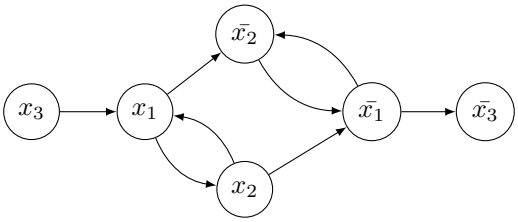
Prenons comme exemple la formule suivante :

$$(x_1 \vee \bar{x_2}) \wedge (\bar{x_1} \vee x_2) \wedge (\bar{x_1} \vee \bar{x_2}) \wedge (x_1 \vee \bar{x_3})$$

Le graphe d'implication sera donc composé des arêtes suivantes :

$$\begin{array}{cccc} \bar{x_1} \Rightarrow \bar{x_2} & x_1 \Rightarrow x_2 & x_1 \Rightarrow \bar{x_2} & \bar{x_1} \Rightarrow \bar{x_3} \\ x_2 \Rightarrow x_1 & \bar{x_2} \Rightarrow \bar{x_1} & x_2 \Rightarrow \bar{x_1} & x_3 \Rightarrow x_1 \end{array}$$

D'où le graphe d'implication :



Si on peut atteindre x_i à partir de $\bar{x_i}$, et si on peut atteindre $\bar{x_i}$ à partir de x_i, alors la formule n'a pas de solution. Pour satisfaire une formule, il faut que x_i et $\bar{x_i}$ soient dans des composantes fortement connexes différentes. Calculons alors les composantes fortement connexes du graphe orienté en utilisant **l'algorithme de Kosaraju**.

1.4 Algorithme de Trajan

L'algorithme de Trajan permet de déterminer les composantes fortement connexes d'un graphe orienté.

1.4.1 Description

On lance un parcours en profondeur depuis un sommet arbitraire. Les sommets explorés sont placés sur une pile P. Un marquage spécifique permet de distinguer certains sommets : les racines des composantes fortement connexes, c'est-à-dire les premiers sommets explorés de chaque composante. Lorsqu'on termine l'exploration d'un sommet racine v, on retire de la pile tous les sommets jusqu'à v inclus. L'ensemble des sommets retirés forme une composante fortement connexe du graphe. S'il reste des sommets non atteints à la fin du parcours, on recommence à partir de l'un d'entre eux.

On s'appuie sur cette algorithme pour déterminer si une formule est satisfaisable.

1.4.2 Implémentassions

```
fonction tarjan(graphe G)
   num := 0
   P := pile vide
   partition := ensemble vide

   fonction parcours(sommet v)
     v.num := num
     v.numAccessible := num
     num := num + 1
     P.push(v), v.dansP := oui

     # Parcours recursif
     pour chaque w successeur de v
       si w.num non defini
         parcours(w)
         v.numAccessible := min(v.numAccessible, w.numAccessible)
       sinon si w.dansP = oui
         v.numAccessible := min(v.numAccessible, w.num)

     si v.numAccessible = v.num
       # v est une racine, on calcule la composante fortement connexe
     associee
       C := ensemble vide
       repeter
         w := P.pop(), w.dansP := non
         ajouter w a C
       tant que w different de v
       ajouter C a partition
   fin de fonction

   pour chaque sommet v de G
     si v.num non defini
       parcours(v)

   renvoyer partition
fin de fonction
```

Listing 1.1 – Algo de Trajan

Chapitre 2

Énoncés

2.1 Exercice 1

Soit F une formule 2-SAT avec n variables. Quel est le nombre maximum de clauses que peut comporter F ?

2.2 Exercice 2

Donnez un exemple de formule 2-SAT qui n'est pas satisfaisable.

2.3 Exercice 3

Écrivez une fonction qui étant donné une affectation pour les variables détermine si la formule est évaluée à vrai ou faux.

2.4 Exercice 4

Étant donné une formule 2-SAT, écrivez une fonction qui construit le graphe orienté associé.

2.5 Exercice 5

A partir du graphe construit à l'exercice précédent, implémentez l'algorithme de [1] qui détermine si la formule est valide ou pas. Si la formule est valide, la fonction renverra une affectation.

Chapitre 3

Résolutions

3.1 Exercice 1

Supposons une formule F avec n variables et m clauses. On souhaite déterminer le nombre de clauses m en fonction du nombre de variable n. Pour le calcule de m, on prend en compte les n variables et leurs compléments $\bar{n}$.

On a donc une combinaison de $2n$ éléments non ordonnée et distinct.

$$F_n = \binom{2n}{2} = \frac{(2n)!}{2!(2n-2)!}$$

Preuve

Sur le nombre n de variables.

Base Soit une formule 2-SAT avec $n = 2$. On a donc le nombre de clause :

$$m = \binom{4}{2} = \frac{4!}{4} = 6$$

Induction Supposons que F est vrai pour n variables, prouvons que F est vrai pour $n + 1$.

$$F_{n+1} = \binom{2(n+1)}{2} = \frac{(2(n+1))!}{2!(2(n+1)-2)!}$$

$$F_{n+1} = \frac{(2n)! \times 2(n+1)}{2!(2n-2)! \times (2(n+1)-2)}$$

$$F_{n+1} = \frac{(2n)!}{2!(2n-2)!} \times \frac{2n+2}{2n+2-2}$$

$$F_{n+1} = F_n \times \frac{n+1}{n}$$

Comme on a $n > 0$, on obtient bien $\binom{2n}{2}$ clauses.

3.2 Exercice 2

Une formule 2-SAT non satisfaisable est une formule qui sera fausse peut importe les valeurs des variables qui la compose.

Soit une formule non satisfaisable :

$$F = (x_1 \vee x_2) \wedge (\bar{x_1} \vee \bar{x_2}) \wedge (x_1 \vee \bar{x_2}) \wedge (\bar{x_1} \vee x_2)$$

x_1	x_2	$x_1 \vee x_2$	$\bar{x_1} \vee \bar{x_2}$	$x_1 \vee \bar{x_2}$	$\bar{x_1} \vee x_2$	F
0	0	0	1	1	1	0
0	1	1	1	0	1	0
1	0	1	1	1	0	0
1	1	1	0	1	1	0

On obtient bien une formule qui vaut **Faux** pour n'importe quel valeurs de x_1 ou x_2.

3.3 Exercice 3

On cherche à créer une fonction d'évaluation pour une formule 2-SAT.

3.3.1 Implémentassions

```
def evaluation( formule ):
    i = 0
    res = 0
    for clause in formule:
        if valeurs.get(clause[0])==None:
            valeurs[clause[0]] = int(not(valeurs.get(clause[0][1:])))
        if valeurs.get(clause[1])==None:
            valeurs[clause[1]] = int(not(valeurs.get(clause[1][1:])))
        formule[i][0] = valeurs.get(clause[0])
        formule[i][1] = valeurs.get(clause[1])
        formule[i] = clause[0] or clause[1]
        i += 1

    for index in range(0,len(formule)-1):
        res = formule[index] and formule[index+1]
    return res


valeurs = {'x1': 1, 'x2': 0, 'x3': 0}
formule = [
        ['x1', '!x2'],
        ['!x1', 'x2'],
        ['!x1', '!x2'],
        ['x1', '!x3']
]
print formule
print evaluation( formule )
```

Listing 3.1 – Fonction evaluation()

3.3.2 Fonctionnement

Soit la formule $F = (x_1 \vee \bar{x_2}) \wedge (x_1 \vee x_3) \wedge (x_2 \vee \bar{x_3}) \wedge (\bar{x_1} \vee \bar{x_2}) \wedge (x_1 \vee x_2)$, avec $x_1 = 1$, $x_2 = 0$ et $x_3 = 0$.

Afin d'évaluer F, on parcourt une première fois la formule. Pour chaque clause, on cherche dans un dictionnaire de donnée si la variable est affectée. Si elle ne l'est pas, on remplace la valeur x par sa valeur inverse $\bar{x}$. Ensuite, on remplace dans les clauses les valeurs de x trouvées dans le dictionnaire de donnée. Pour terminer le premier parcours, on évalue individuellement la valeur de chacune des clauses en appliquant le "ou" logique.

Enfin, on parcourt la formule une seconde fois en appliquant le "et" logique entre chaque clause. Puis on retourne le résultat.

3.4 Exercice 4

On cherche à créer une fonction qui créer un graphe orienté à partir d'une fromule 2-SAT.

3.4.1 Implémentassions

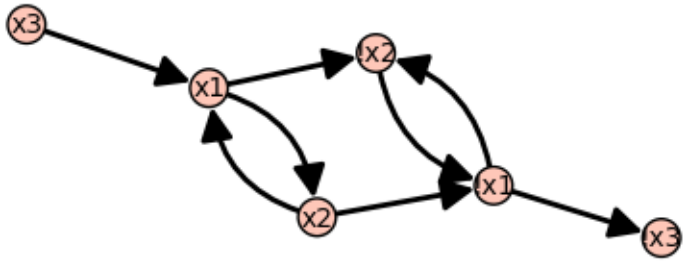
```
def graphe_oriente( formule ):
    aretes = []
    graphe = DiGraph()
    for clause in formule:
        aretes.append(["!" + clause[0], clause[1]])
        aretes.append(["!" + clause[1], clause[0]])
    for arete in aretes:
        if arete[0][:2]=="!!":
            arete[0] = arete[0][2:]
        if arete[1][:2]=="!!":
            arete[1] = arete[1][2:]
    graphe.add_edges(aretes)
    return graphe

valeurs = {'x1': 1, 'x2': 0, 'x3': 0}
formule = [['x1','!x2'],['!x1','x2'],['!x1','!x2'],['x1','!x3']]
G = graphe_oriente( formule )
G.show()
```

Listing 3.2 – Fonction graphe Orienté

3.4.2 Fonctionnement

Soit F une formule 2-SAT. Pour chaque clause de F, on ajoute à la liste d'arêtes $\bar{x_i} \Rightarrow x_j$ et $\bar{x_j} \Rightarrow x_i$. Comme on manipule des chaîne de caractères, on s'assure ensuite d'enlever les répétitions de " !" . Puis on ajoute les arêtes au graphe et on l'affiche.

3.4.3 Résultats



On obtient le résultat espéré dans dans la partie 1.3 du rapport.

3.5 Exercice 5

3.5.1 Implémentassions

```
def pp( graphe ):
    for noeud in graphe:
        visite[noeud] = "blanc"
    for noeud in graphe:
        if visite[noeud] == "blanc":
            visiterpp( noeud )

def visiterpp( noeud ):
    visite[noeud] = "gris"
    if len(graphe.outgoing_edges([noeud])) > 0:
        for v in graphe.outgoing_edges([noeud]):
            if visite[v[1]] == "blanc":
                visite[v[1]] = "gris"
                visiterpp(v[1])
    visite[noeud] = "noir"
    etape.append(noeud)

def reverse( graphe ):
    D = DiGraph()
    for v in graphe.edges():
        D.add_edge(v[1],v[0])
    return D

def trajan( graphe ):
    visite = {}
    etape = []
    pp( graphe )
    etape.reverse()
    graphe = reverse( graphe)
    return graphe
valeurs = {'x1': 1, 'x2': 0, 'x3': 0}
formule = [['x1','!x2'],['!x1','x2'],['!x1','!x2'],['x1','!x3']]

G = graphe_oriente( formule )
trajan( G )
```

Listing 3.3 – Fonction Trajan

3.5.2 Fonctionnement

On s'appuie sur l'algorithme de Trajan pour déterminer si la formule est valide. Pour ce faire, on effectue un parcours en profondeur en partant du graphe issues de la formule. On note l'ordre typologique lors dur parcours. Une fois le graphe parcourus, on inverse la liste de l'ordre, puis on effectue un DFS sur le graphe inverse de G en suivant l'ordre de la nouvelle liste. Une fois terminé, on obtient toutes les composantes fortement connexes du graphe. On vérifie alors si pour tout x_i, $\bar{x_i}$ appartient à la même composante fortement connexe, et vis versa.

Si on obtient **Faux**, la formule n'est pas satisfaisable, sinon, on cherche une solution et on la renvoie.

Chapitre 4

Remarques

4.1 Problèmes rencontrés

4.1.1 Gestion de la charge de travail

Le problème le plus évident rencontré lors de ce travail à été la gestion de la charge de travail. Un manque de communication à entraîné un certain quiproquo sur la répartition des tâches, ce qui à nécessairement provoqué de la perte de temps effectif.

4.1.2 Difficulté des algorithme

Nous avons rencontrés quelques difficultés au niveau de l'algorithmique. Nous avons eu du mal à déterminer un algorithme qui fonction quelque soit le nombre de variable que comporte notre formule, ou même l'implémentation de l'algorithme de **Trajan**.

4.2 Conclusion générale

Nous n'avons pas vraiment eu de problèmes de compréhensions des algorithmes, mais plutôt d'implantation, **python** et **SageMath** étant nouveau pour nous.

Bibliographie

[1] Algorithms live! - episode 24 - 2sat. https://www.youtube.com/watch?v=0nNYy3rltgA.

[2] Cp-algorithms, 2 - sat. https://cp-algorithms.com/graph/2SAT.html.

[3] Rosetta code. https://rosettacode.org/wiki/Kosaraju#Python.

[4] Wiki trajan. https://cutt.ly/8hgBHaP.