INTRODUCTION

Quand les scientifiques ont voulu énoncer formellement et rigoureusement ce qu'est l'efficacité d'un algorithme ou au contraire sa complexité, ils se sont rendu compte que la comparaison des algorithmes entre eux était nécessaire et que les outils pour le faire à l'époque [1] étaient primitifs. Dans le but d'analyser et de comparer des algorithmes, on utilise souvent l'approche dite du "pire cas". Cette approche consiste en gros à évaluer comment le temps d'exécution d'un algorithme est relié à la taille de la donnée d'entrée (*input*). Dans ce type de calcul, on ignore généralement certains facteurs constants qui ne peuvent être contrôlés par le programmeur tels que la traduction en langage machine du programme dans lequel l'algorithme est écrit, le type d'ordinateur sur lequel le programme va rouler, etc... De plus on s'intéresse davantage à évaluer la performance de l'algorithme pour les grandes tailles d'entrée que pour les petites. Tous les algorithmes étudiés jusqu'ici étaient des ***algorithmes à temps polynomial*** : sur des entrées de taille n, leur temps d'exécution dans le pire des cas est $O(n^k)$ pour une certaine constante k. Il est naturel de se demander si tous les problèmes peuvent être résolus en temps polynomial. La réponse est non. Par exemple, il existe des problèmes, tel le fameux « problème de l'arrêt de Turing d'une machine », qui ne peuvent être résolus par aucun ordinateur, quel que soit le temps qu'il y passe. Il existe aussi des problèmes qui peuvent être résolus, mais pas en temps $O(n^k)$ pour une constante k quelconque. De manière générale, on considère que les problèmes résolubles par des algorithmes à temps polynomial sont traitables (faciles) et que les problèmes nécessitant un temps supra-polynomial (dits NP-Complets) sont intraitables (difficiles). Dans ce qui suit, on présentera tout d'abord l'historique de la NP-complétude, les classes de complexité *P, NP et NP-complet*, ensuite on présentera la théorie de la NP-COMPLETUDE et la réduction polynomiale et enfin quelques problèmes et montrer qu'ils sont *NP*-complet.

I. HISTORIQUE

Le concept de *NP*-complétude a été introduit en 1971 par **Stephen Cook** dans une communication intitulée ***The complexity of theorem-proving procedures*** (La complexité des procédures de démonstration de problèmes), bien que le mot « NP-complet » n'apparaisse pas explicitement dans l'article. Lors de la conférence à laquelle il a été présenté, une discussion acharnée a eu lieu entre les chercheurs présents pour savoir si les problèmes *NP*-complets pouvaient être résolus en temps polynomial sur machine de Turing déterministe. **John Hopcroft** a finalement convaincu les participants que la question devait être remise à plus tard, personne n'ayant réussi à démontrer ou infirmer le résultat.

La question de savoir si ***P = NP*** n'est toujours pas résolue. Elle fait partie des problèmes du prix du millénaire, un ensemble de sept problèmes pour lesquels l'Institut de mathématiques Clay offre un prix d'un million de dollars. Il « suffirait » de trouver un seul problème *NP* qui soit à la fois *NP*-complet et *P* pour démontrer cette hypothèse, ou d'exhiber un seul problème *NP* qui ne soit pas dans P pour démontrer sa négation.

Le résultat de l'article de Cook, démontré de manière indépendante par Leonid Levin en URSS, est maintenant connu sous le nom de **théorème de Cook-Levin.** Ce théorème affirme qu'il existe un problème NP-complet. Cook a choisi le problème SAT et Levin un problème de pavage. En 1972, **Richard Karp** a prouvé la NP-complétude de plusieurs autres problèmes fondamentaux en informatique et très disparates, connus comme la liste des 21 problèmes NP-complets de Karp. Depuis, on a démontré la NP-complétude des milliers d'autres problèmes.

II. PRESENTATION DES CLASSES

Un problème de décision est un problème dont la réponse est OUI ou NON/ VRAI ou FAUX (il pleut ? 3 est un nombre pair ?). Pour un problème P, notons *O*(P) l'ensemble des instances de P dont la réponse est OUI. Ainsi selon la difficulté à résoudre efficacement un problème et à vérifier la véracité de sa solution, on distingue :

1. La classe P

La classe de complexité P ou classe Polynomiale se compose des problèmes qui sont résolubles en temps polynomial par rapport à la taille de l'entrée. Autrement dit cette classe regroupe les problèmes qui sont résolubles en temps $\boldsymbol{O(n^k)}$ pour une certaine constance k (n est la taille des entrées) . C'est la classe des problèmes dits faciles. On dit aussi, pour abréger, que l'algorithme lui-même est polynomial. Les algorithmes efficaces sont polynomiaux, c'est à dire les algorithmes non polynomiaux sont certainement inefficaces. A titre d'exemple, nous pouvons citer : le problème du plus court chemin (Dijkstra) et le problème de la détermination de l'arbre de recouvrement de poids minimum (Prim et Kruskal).

2. La classe NP

La classe NP se compose des problèmes qui sont « vérifiables » en temps polynomial. Cela signifie : si l'on nous donne, d'une façon ou d'une autre, une solution « certifiée », nous pouvons vérifier que cette solution est correcte en temps polynomial par rapport à la taille de l'entrée. Par exemple, dans le problème du circuit hamiltonien, étant donné un graphe orienté $\boldsymbol{G = (S, A)}$, une solution certifiée serait une suite $\boldsymbol{(}v_1, v_2, v_3,..., v_{|S|}\boldsymbol{)}$ de|S| sommets. Il est facile de vérifier en temps polynomial que $(v_i, v_{i+1}) \in A$ pour i =1, 2,3,..., |S|−1 et que $(v_{|S|}, v1) \in A$ aussi.

Théorème : $\boldsymbol{P \subseteq NP}$

Preuve : Soit p ∈ P et soit A un algorithme polynomial permettant de résoudre chaque instance de P. Pour une instance $p \in O(P)$, on peut vérifier que $p \in O(P)$ en appliquant A, qui donne la réponse OUI en un temps0 polynomial. Ceci prouve que $\boldsymbol{P \subseteq NP}$.

3. La classe NP-Complet

La classe NP-complet contient les problèmes les plus difficiles de **NP**. Ces problèmes semblent vraiment ne pas faire partie de P (bien que à l'heure actuelle on ne possède pas une réponse définitive, voir la question irrésolue $\boldsymbol{P = NP ?}$). Si nous sommes capables de trouver un moyen de résoudre efficacement (algorithme polynomial) un problème de NP-complet, nous pouvons alors utiliser cet algorithme pour résoudre tous les problèmes de NP.

En effet, un problème *X* appartient à NP-complet s'il est dans NP et si tout autre problème dans NP peut s'y réduire. On n'en déduit qu'une méthode "assez facile" pour prouver qu'un nouveau problème appartient à NP-complet est de montrer d'abord qu'il est dans NP, ensuite le réduire en un problème connu dans NP-complet. Par conséquent il est très utile de connaître une variété de problèmes NP-complets.

III. THEORIE DE LA NP-COMPLETUDE

1) Définition et principe

En théorie de la complexité, un problème NP-complet ou problème NPC (c'est-à-dire un problème complet pour la classe NP) est un problème de décision vérifiant les propriétés suivantes :

-il est possible de vérifier une solution efficacement (en temps polynomial) ; la classe des problèmes vérifiant cette propriété est notée NP ;
-tous les problèmes de la classe NP se ramènent à celui-ci via une réduction polynomiale ; cela signifie que le problème est au moins aussi difficile que tous les autres problèmes de la classe NP.

Un problème NP-difficile est un problème qui remplit la seconde condition, et donc peut être dans une classe de problème plus large et donc plus difficile que la classe NP.

Bien qu'on puisse vérifier rapidement toute solution proposée d'un problème NP-complet, on ne sait pas en trouver efficacement. C'est le cas, par exemple, du problème du voyageur de commerce ou de celui du problème du sac à dos.

Tous les algorithmes connus pour résoudre des problèmes NP-complets ont un temps d'exécution exponentiel en la taille des données d'entrée dans le pire des cas, et sont donc inexploitables en pratique même pour des instances de taille modérée.

La seconde propriété de la définition implique que s'il existe un algorithme polynomial pour résoudre un quelconque des problèmes NP-complets, alors tous les problèmes de la classe NP peuvent être résolus en temps polynomial. Trouver un algorithme polynomial pour un problème NP-complet ou prouver qu'il n'en existe pas permettrait de savoir si P = NP ou P

≠ NP, une question ouverte qui fait partie des problèmes non résolus en mathématiques les plus importants à ce jour.

En pratique, les informaticiens et les développeurs sont souvent confrontés à des problèmes NP-complets. Dans ce cas, savoir que le problème sur lequel on travaille est NP-complet est une indication du fait que le problème est difficile à résoudre, donc qu'il vaut mieux chercher des solutions approchées en utilisant des algorithmes d'approximation ou utiliser des heuristiques pour trouver des solutions exactes

La raison qui pousse le plus les théoriciens de l'informatique à croire que P ≠ NP est sans doute l'existence de la classe des problèmes «NP-complets ». Cette classe possède une propriété surprenante : si un problème NP-complet peut être résolu en temps polynomial, alors tous les problèmes de NP peuvent être résolus en temps polynomial, autrement dit, P = NP. Pourtant, malgré des années de recherches, aucun algorithme polynomial n'a jamais été découvert pour quelque problème NP-complet que ce soit.

Les langages NP-complets sont, dans un sens, les langages les plus « difficiles » de NP. La réductibilité en temps polynomial va donc nous permettre de comparer des problèmes.

2) Réduction polynomiale

Cette notion permettant de montrer qu'un problème n'est pas plus difficile ou plus facile qu'un autre, s'applique même quand les deux problèmes sont des problèmes de décision. Nous exploiterons cette idée dans la quasi-totalité des démonstrations de NP-complétude, et ce de la façon suivante. Soit un problème de décision, disons *A*, que l'on voudrait résoudre en temps polynomial. L'entrée d'un problème particulier est dite ***instance*** de ce problème ; Supposons maintenant qu'il y ait un autre problème de décision, disons ***B***, que nous savons déjà comment résoudre en temps polynomial. Enfin, supposons que nous ayons une procédure qui transforme toute instance a de *A* en une certaine instance b de *B* et qui a les caractéristiques suivantes :

- La transformation prend un temps polynomial.
- Les réponses sont les mêmes. C'est à dire, la réponse pour **a** est « **oui** » si et seulement si la réponse pour **b** est « **oui** ».

Une telle procédure porte le nom de ***réduction*** à temps polynomial et, comme le montre la figure si dessous, elle donne un moyen de résoudre le problème ***A*** en temps polynomial : Étant donnée une

- Instance **a** de ***A***, utiliser une réduction à temps polynomial pour la transformer en une instance **b** de ***B***.
- Exécuter l'algorithme de décision à temps polynomial de ***B*** sur l'instance **b**.
- Utiliser la réponse pour **b** comme réponse pour **a**.
- Si chacune de ces étapes prend un temps polynomial, il en est de même de l'ensemble ; on a donc un moyen de décider pour a en temps polynomial. Autrement dit, en « réduisant » la résolution du problème *A* à celle du problème *B*, on se sert de la « facilité » de *B* pour prouver la « facilité » de *A*.

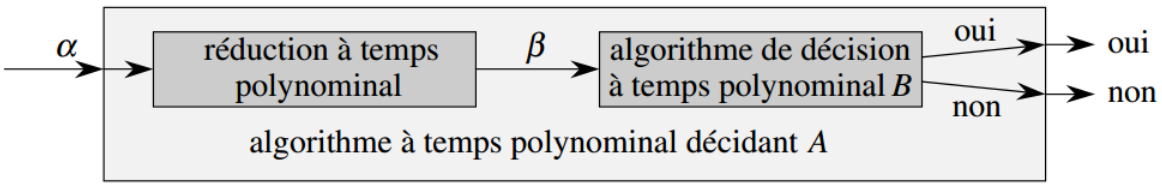

Figure 1 : *Utilisation d'une réduction à temps polynomial pour résoudre un problème de décision A (α) en temps polynomial à partir d'un algorithme de décision à temps polynomial associé à un autre problème B (β). En temps polynomial, on transforme une instance a de A en une instance b de B, on résout B en temps polynomial, puis on utilise la réponse pour b comme réponse pour a.*

En nous rappelant que la NP-complétude consiste à démontrer le niveau de difficulté d'un problème et non son niveau de facilité, nous pouvons utiliser des réductions à temps polynomial, en sens inverse, Pour prouver qu'un problème est NP-complet. Poussons l'idée un cran plus loin et montrons comment nous pourrions employer des réductions à temps polynomial pour prouver qu'il ne peut exister d'algorithme à temps polynomial pour un certain problème *B*. Supposons que l'on ait un problème de décision *A* pour lequel on sait déjà qu'il ne peut pas exister d'algorithme à temps polynomial. (Nous laisserons de côté, pour l'instant, le problème de savoir comment trouver un tel problème *A*.) Supposons, en outre, que l'on ait une réduction à temps polynomial qui transforme des instances de *A* en instances de *B*. On peut alors utiliser un raisonnement simple par l'absurde pour prouver qu'il ne peut pas exister d'algorithme à temps polynomial pour *B*. Supposons le contraire, c'est à dire que *B* ait un algorithme à temps polynomial. Alors, en utilisant la méthode illustrée à la ***figure1***, on aurait un moyen de résoudre *A* en temps polynomial, ce qui est contraire à l'hypothèse selon

laquelle il n'y a pas d'algorithme à temps polynomial pour *A*. Pour la NP-complétude, on ne peut pas partir du principe qu'il n'y a absolument pas d'algorithme à temps polynomial pour le problème *A*. Cependant, la méthodologie de la démonstration est la même : nous prouvons que le problème *B* est NP-complet en supposant que le problème *A* est lui aussi NP-complet.

Définition :

Réduction polynomiale

Soient A et B deux problèmes de décision. On dit que A se réduit polynomialement à B et on note $A \leq p\ B$ si il existe une fonction $f : \sum * \rightarrow \sum *$ calculable en temps polynomial telle que $x \in A\ ssi\ f(x) \in B$.

NP-complet

A est NP-Complet si :

- $\boldsymbol{A \in NP}$
- A est aussi difficile que n'importe quel problème NP c'est-à-dire $\boldsymbol{B \leq_P A}, B \in NP$

Tout problème P ∈ NP ; en effet, si un problème appartient à P alors on peut le résoudre en temps polynomial même si l'on ne nous donne pas de solution ; d'où la relation $\boldsymbol{P \subseteq NP}$; Mais on n'a pas encore put prouver que **NP = P** car bien qu'on puisse vérifier en temps polynomiale toute solutions proposées, on ne sait pas en trouver efficacement. Ainsi la classe NP regroupe la classe NP et la classe NPC (voir la figure ci-dessous).

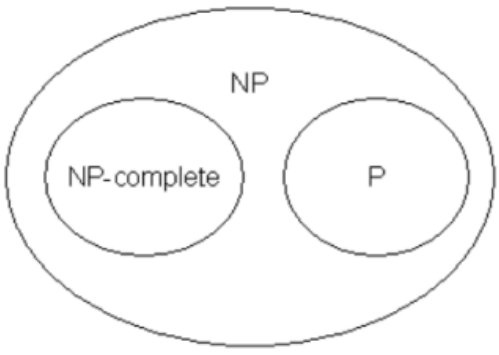

Figure 2 : Structure de classe

IV. QUELQUES PROBLEMES NP-COMPLETS

i. Satisfaisabilité d'un circuit

Un circuit est dit satisfiable s'il existe une assignation aux valeurs d'entrées telles que le circuit global produit « OUI » ou « 1 ».
Pour montrer que ***CUIRCUIT-SAT est NP-complet***, nous allons d'abord :

a- Montrons que *CUIRCUIT-SAT* ϵ *NP*

Soit C un circuit constitué de n portes logiques et A une assignation qui satisfait le circuit. Montrons que la vérification de A sur C se fait en temps polynomial.

C est constitué de n portes logiques, la vérification de A sur C implique que nous allons évaluer les résultats de ces n portes logiques en fonction de leurs entrées. D'où la vérification de A sur C se fait dans l'ordre n portes.
$O(n)$ est un temps polynomial. D'où *CUIRCUIT-SAT* ϵ *NP* NP

b- Montrons que *CUIRCUIT-SAT* est *NP-difficile (complexité exponentielle)*

Etant donné que *CUIRCUIT-SAT* est satisfaisable, on regarde simplement la porte qui donne le résultat global du circuit, on exprime de manière récurrente chacune des entrées en tant que formule. La formule correspondante au circuit est alors obtenue en écrivant une expression qui applique la fonction de la porte aux formules de ces entrées.
Malheureusement cette méthode directe n'est pas une résolution en temps polynomial.
Les sous formules partagées peuvent faire croitre exponentiellement la taille de la formule générée pour peu qu'il ait des portes donc les files de sorties est de facteur d'éventail de 2 ou plus (croit de façon exponentielle). D'où la complexité dans le cas est de l'ordre d'exponentielle.

D'où *CUIRCUIT-SAT* est *NP-difficile*.

Nous avons montré que *CUIRCUIT-SAT* ϵ NP d'après a, et d'après b nous avons montré qu'il est *NP-difficile*. D'où ***CUIRCUIT-SAT*** **est** *NP-complet***.**

ii) Satisfaisabilité d'une formule booléenne

En informatique théorique, le problème SAT ou problème de satisfaisabilité booléenne est le problème de décision, qui, étant donnée une formule de logique propositionnelle, détermine s'il existe une assignation des variables propositionnelles qui rend la formule vraie. Ce problème eut l'honneur d'être le premier dans l'histoire dont on a démontré qu'il était NP-complet. Nous exprimons le problème de la satisfaisabilité (de formule) en fonction du langage SAT de la manière suivante.

Une instance de SAT est une formule booléenne f composée de

- n variables booléennes : $x_1 , x_2, \ldots, x_n$;
- m connecteurs booléens : toute fonction booléenne avec une ou deux entrées et une sortie, telle que $\wedge\ (ET), \vee\ (OU), \neg\ (NON), \rightarrow\ (implication), \leftrightarrow\ (si\ et\ seulement\ si)$; et
- des parenthèses. (Sans nuire à la généralité, on suppose qu'il n'y a pas de parenthèses redondantes, c'est à dire qu'il y a au plus une paire de parenthèses par connecteur booléen.)

Il est facile d'écrire une formule booléenne f dans une longueur qui est polynomiale en $n + m$. Comme c'est le cas avec les circuits combinatoires booléens, une assignation de vérité pour une formule booléenne f est un ensemble de valeurs pour les variables de f, et une assignation satisfaisante est une assignation pour laquelle l'évaluation de la formule donne 1. Une formule possédant une assignation satisfaisante est une formule satisfaisable. Le problème de la satisfaisabilité consiste à savoir si une formule booléenne donnée est satisfaisable ; en termes des langages formels, on dira

$$\boldsymbol{SAT} = \{\langle\phi\rangle \boldsymbol{tel\ que\ est\ une\ formule\ bool\acute{e}enne\ satisfaisable}\}.$$

Exemple : La formule

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

possède l'assignation satisfaisante$\langle x_1 = 0 , x_2 = 0 , x_3 = 1 , x_4 = 1 \rangle$, puisque

$$\phi = \Big((0 \rightarrow 0) \vee \neg\big((\neg 0 \leftrightarrow 1) \vee 1\big)\Big) \wedge \neg 0$$

$$= (1 \vee \neg(1 \vee 1)) \wedge 1$$

$$= (1 \vee 0) \wedge 1$$

$$= 1$$

et donc cette formule ϕ appartient à SAT.

L'algorithme naïf permettant de déterminer si une formule booléenne arbitraire est satisfaisable ne s'exécute pas en temps polynomial. Il existe 2^n assignations possibles d'une formule ϕ à n variables. Si la longueur de f est polynomiale en n, alors la vérification de chaque assignation demande un temps $\Omega(2^n)$, ce qui est suprapolynomial par rapport à la longueur de ϕ. Comme nous allons le montrer, un algorithme polynomial a très de chance d'exister .

Pour montrer que La satisfaisabilité d'une formule booléenne est un problème ***NP-Complet.*** Nous allons :

a- $\boldsymbol{SAT \in NP}$

On montre qu'un certificat consistant en une assignation satisfaisante pour une formule f en entrée peut être validé en temps polynomial.

L'algorithme de vérification remplace tout simplement chaque variable de la formule par sa valeur correspondante, puis évalue l'expression.

Ce travail peut facilement s'effectuer en temps polynomial. Si l'expression a pour évaluation 1, la formule est satisfaisable.

Donc, $\boldsymbol{SAT \in NP}$

b- $\boldsymbol{SAT}$ **est** ***NP-difficile***

On montre que $\boldsymbol{CIRCUIT\text{-}SAT \leq_P SAT}$.

Autrement dit, toute instance du problème de satisfaisabilité de circuit peut être réduite en temps polynomial à une instance du problème de satisfaisabilité de formule. On peut faire

appel à une récurrence pour exprimer un circuit combinatoire booléen quelconque comme une formule booléenne.

On regarde simplement la porte qui produit le résultat du circuit, et on exprime de manière récurrente chacune des entrées de la porte en tant que formules. La formule correspondant au circuit est alors obtenue en écrivant une expression qui applique la fonction de la porte aux formules de ses entrées.

Malheureusement, **cette méthode directe n'est pas une réduction en temps polynomial**. Les sous-formules partagées peuvent faire croître exponentiellement la taille de la formule générée, pour peu qu'il y ait des portes dont les fils de sortie aient des facteurs d'éventail de 2 ou plus.

Donc, l'algorithme de réduction devra être un peu plus astucieux.

La figure 4 illustre le principe fondamental de la réduction de CIRCUIT-SAT à SAT Pour chaque fil *xi* du circuit *C*, la formule ϕ à une variable *xi*. L'opération propre à une porte peut maintenant être exprimée comme une formule mettant en jeu les variables situées de ses fils incidents.

Par exemple, l'opération de la porte ET en sortie est $\boldsymbol{x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)}$.

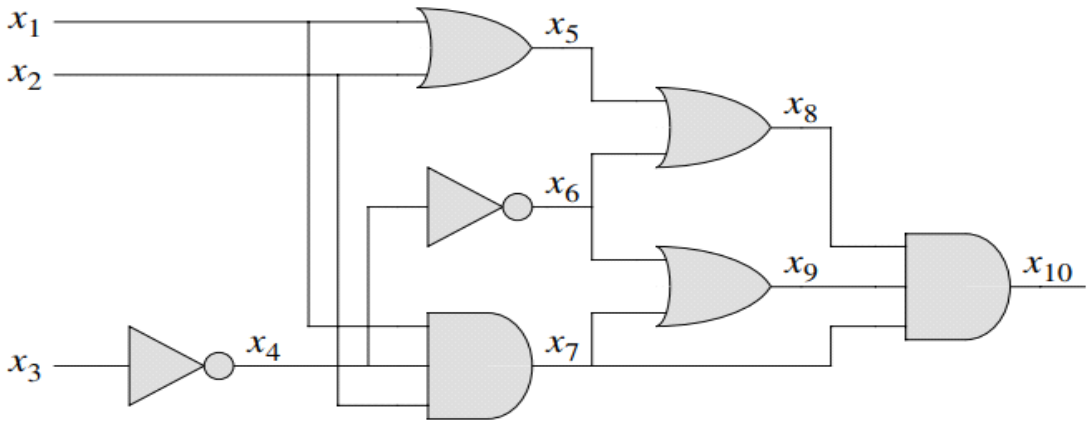

Figure 4 : *Réduction du problème de la satisfaisabilité de circuit à celui de la satisfaisabilité de formule. La formule produite par l'algorithme de réduction a une variable pour chaque fil du circuit.*

La formule ϕ produite par l'algorithme de réduction est **le ET entre la variable de sortie du circuit et la conjonction des clauses décrivant l'opération de chaque porte**.

Pour le circuit de la figure 4, la formule est :

$$\phi = x_{10} \wedge (x_4 \leftrightarrow \neg x_3)$$
$$\wedge (x_5 \leftrightarrow (x_1 \vee x_2))$$
$$\wedge (x_6 \leftrightarrow \neg x_4)$$
$$\wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4))$$
$$\wedge \left(x_8 \leftrightarrow (x_5 \vee x_6)\right)$$
$$\wedge (x_9 \leftrightarrow (x_6 \vee x_7))$$
$$\wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)) .$$

Étant donné un circuit *C*, il est immédiat de produire une telle formule f en temps polynomial.

Pourquoi le circuit C est-il satisfaisable exactement quand la formule f est satisfaisable ?

Si *C* possède une assignation satisfaisante, chaque fil du circuit a une valeur bien définie et la sortie du circuit est 1. Donc, l'assignation de valeurs de fil aux variables de f fait que chaque clause de f est évaluée à 1, et donc que leur conjonction est aussi évaluée à 1. **Réciproquemen**t, s'il existe une assignation qui donne une évaluation 1 pour f, le circuit *C* est satisfaisable de par une démonstration analogue. On a donc montré que ***CIRCUIT-SAT*** $\leq_P$ ***SAT***.

Donc ***SAT*****est un problème *NP-Complet*.**

iii) Satisfaisabilité *3-CNF*

On peut démontrer la NP-complétude de nombreux problèmes par réduction à partir la satisfaisabilité de formule. L'algorithme de réduction doit cependant gérer toute formule donnée en entrée, ce qui peut amener à considérer un nombre de cas très important. Il est donc souvent souhaitable de réduire à partir d'un langage restreint de formules booléennes, de façon à considérer moins de cas. Bien sûr, il ne faut pas restreindre le langage au point de le rendre résoluble en temps polynomial. Un langage commode est celui de la satisfaisabilité *3-CNF*, ou *3-CNF-SAT*.

On définit la satisfaisabilité *3-CNF* à l'aide des termes suivants. Un littéral dans une formule booléenne est une variable ou sa négation. Une formule booléenne se trouve dans sa forme normale conjonctive, ou forme CNF, si elle est exprimée comme un ET de clauses, chacune d'elles étant le OU d'un ou plusieurs littéraux. Une formule booléenne se trouve dans sa forme normale conjonctive d'ordre 3, ou *3-CNF*, si chaque clause comporte exactement

trois littéraux distincts. Par exemple, la formule booléenne est en forme *3-CNF*. La première de ses trois clauses est qui contient les trois littéraux.

Dans *3-CNF-SAT,* on cherche à savoir si une formule booléenne f donnée en forme *3-CNF* est satisfaisable. La démonstration qui suit, montre qu'un algorithme polynomial qui peut déterminer la satisfaisabilité d'une formule booléenne a très peu de chances d'exister, même quand elle est exprimée dans cette forme normale simple.

Pour montrer Le problème de la satisfaisabilité des formules booléennes sous forme ***3-CNF* est *NP-complet***. Nous allons :

a- $\boldsymbol{3\text{-}CNF\text{-}SAT \in NP}$

La démonstration utilisée pour montrer que $SAT \in NP$ s'applique aussi pour montrer que *3-CNF-SAT* $\in NP$.

b- **Montrons que $\boldsymbol{SAT \leq_P}$ *3-CNF-SAT***

L'algorithme de réduction se divise en trois grandes parties. Chaque étape transforme progressivement la formule donnée en entrée f en la forme normale conjonctive d'ordre 3.

- La première étape est similaire à celle utilisée pour démontrer *CIRCUIT-SAT* $\leq_P$ SAT.

On commence par construire ***un arbre binaire « d'analyse »*** pour la formule ϕ, en plaçant les littéraux sur les feuilles et les connecteurs sur les nœuds internes. La figure 5 montre ce type d'arbre d'analyse pour la formule$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$.

Si la formule d'entrée contient une clause comme le OU de plusieurs littéraux, on peut utiliser l'associativité pour parenthèser l'expression complètement de sorte que tout nœud interne de l'arbre résultant possède 1 ou 2 enfants. L'arbre binaire d'analyse peut maintenant être vu comme un circuit pour le calcul de la fonction.

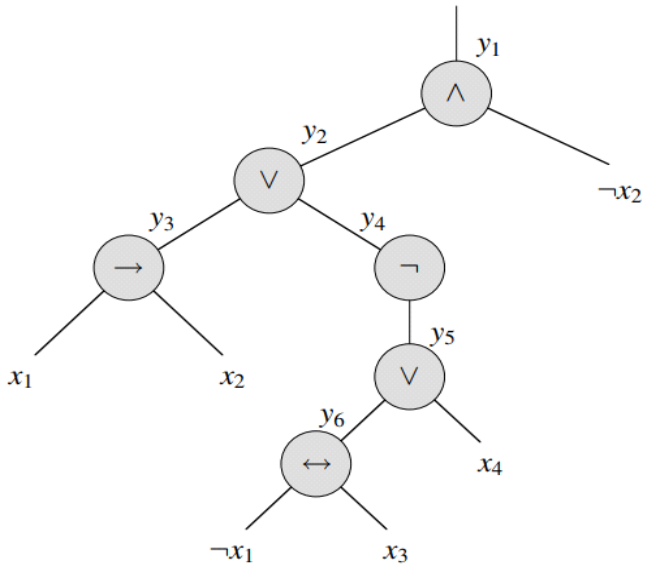

Figure 5 **:** *L'arbre correspondant à la formule* $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$.

une variable *yi* pour la sortie de chaque nœud interne. Ensuite, on réécrit la formule originale ϕcomme le ET entre la variable racine et une conjonction de clauses décrivant l'action de chaque nœud. Pour la formule l'expression qui en résulte est

$$\begin{aligned}\phi' = {} & y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\ & \wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\ & \wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\ & \wedge (y_4 \leftrightarrow \neg y_5) \\ & \wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\ & \wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3)) .\end{aligned}$$

Observez que la formule ϕ'ainsi obtenue est une conjonction de clauses ϕ'_i possédant chacune au plus 3 littéraux. La seule contrainte supplémentaire est que chaque clause soit un OU de littéraux.

➢ La deuxième étape de la réduction convertit chaque clause ϕ'_idans sa forme normale conjonctive. On construit une table de vérité pour ϕ'_ien évaluant toutes les assignations possibles à ses variables. Chaque ligne de la table de vérité représente une assignation possible des variables de la clause, et la valeur de la clause pour cette assignation. En utilisant les éléments de la table de vérité qui sont évalués à 0, on construit une formule en ***forme normale disjonctive*** (ou ***DNF***), c'est à dire un OU de ET, qui est équivalente à $\neg\phi'_i$. Cette formule est ensuite convertie en une formule CNF $\boldsymbol{\phi''_i}$, en faisant appel aux ***lois de Morgan*** pour complémenter tous les littéraux et changer les OU en ET et les ET en OU.

Dans notre exemple, on convertit la clause

$\phi'_1 = (y_1 \leftrightarrow (y_2 \wedge \neg x_2))$ en forme CNF de la façon suivante.

La table de vérité pour ϕ'_1 est donnée à la figure 6. La formule DNF équivalente à$\neg\phi'_1$ *est :*

$$(y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2).$$

En appliquant les lois de DeMorgan, on obtient la formule CNF : $\boldsymbol{\phi''_i} = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2) \wedge (y_1 \vee \neg y_2 \vee x_2)$,

qui est équivalente à la clause initiale $\boldsymbol{\phi'_i}$.

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

Figure 6: *La table de vérité de la clause* $(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$.

Chaque clause $\boldsymbol{\phi'_i}$de la formule $\boldsymbol{\phi'}$est maintenant convertie en une formule CNF $\boldsymbol{\phi''_i}$,et $\boldsymbol{\phi'}$est donc équivalente à la formule CNF $\boldsymbol{\phi''}$composée de la conjonction des $\boldsymbol{\phi''_i}$.Par ailleurs, chaque clause de $\boldsymbol{\phi''}$possède au plus 3 littéraux.

➢ La troisième et dernière étape de la réduction transforme encore la formule, de façon que chaque clause contienne *exactement* 3 littéraux distincts. La formule *3-CNF* finale $\boldsymbol{\phi'''}$est construite à partir des clauses de la formule *NF* $\boldsymbol{\phi''}$. Elle utilise également deux variables auxiliaires que nous appellerons *p* et *q*. Pour chaque clause *Ci* de $\boldsymbol{\phi''}$, on inclut les clauses suivantes dans $\boldsymbol{\phi'''}$:

- Si C_i a 3 littéraux distincts, alors on se contente d'inclure C_i comme clause de f'".

- Si C_i a 2 littéraux distincts, autrement dit si $C_i = (l1 \vee l2)$, où $l1$ et $l2$ sont des littéraux, on inclut dans $\boldsymbol{\phi'''}$,les clauses $(l1 \vee l2 \vee p) \wedge (l1 \vee l2 \vee \neg p)$. Les littéraux p et $\neg p$ servent simplement à respecter la contrainte syntaxique qui veut qu'il y ait exactement 3 littéraux distincts par clause : $(l1 \vee l2 \vee p) \wedge (l1 \vee l2 \vee \neg p)$ est équivalent à $(l1 \vee l2)$, aussi bien pour $p = 0$ que pour $p = 1$.
- Si C_i ne possède qu'un seul littéral l, alors on inclut dans $\boldsymbol{\phi'''}$,les clauses$(l \vee p \vee q) \wedge (l \vee p \vee \neg q) \wedge (l \vee \neg p \vee q) \wedge (l \vee \neg p \vee \neg q)$. Notez que, quelles que soient les valeurs de *p* et *q*, la conjonction de ces quatre clauses donne une évaluation de *l*.

En se penchant sur chacune de ces trois étapes, on peut voir que la ***formule 3-CNF $\boldsymbol{\phi'''}$, est satisfaisable si et seulement si $\boldsymbol{\phi}$,est satisfaisable***.

Comme pour la réduction de *CIRCUIT-SAT* à SAT, la construction de $\boldsymbol{\phi'}$à partir de $\boldsymbol{\phi}$ dans la première étape conserve la satisfaisabilité. La deuxième étape produit une formule CNF $\boldsymbol{\phi''}$qui est algébriquement équivalente à $\boldsymbol{\phi'}$, de même pour la troisième étape qui produit la formule *3-CNF* $\boldsymbol{\phi''}$.

D'où $SAT \leq_P$ *3-CNF-SAT*.

Par conséquent le problème de la satisfaisabilité des formules booléennes sous ***forme 3-CNF*** est ***NP-complet***.

iv) Problème de clique

Une ***clique*** dans un graphe non orienté $\boldsymbol{G = (S, A)}$ est un sous-graphe complet de *G* c'est-à-dire que deux sommets quelconques de la clique sont toujours adjacents. La ***taille*** d'une clique est le nombre de sommets qu'elle contient. Le ***problème de la clique*** est le problème d'optimisation consistant à trouver une clique de taille maximale dans un graphe. Le problème de décision associé est celui de savoir s'il existe une clique de taille $\boldsymbol{K}$ dans le graphe. De manière formelle la définition d'une clique est:

$$\boldsymbol{CLIQUE} = \{< G, k > / \; G \; est \; un \; graphe \; contenant \; une \; Clique \; de \; taille \; k\}$$

- En entrée : Graphe $\boldsymbol{G} = (\boldsymbol{S}, \boldsymbol{A})$et une constante **k**
- Sortie : exit-il dans **G** une clique $\boldsymbol{S'} \subseteq \boldsymbol{S}$avec$|\boldsymbol{S'}| = \boldsymbol{k}$ **?**

Montrer que ***CLIQUE*** est ***NP-Complet***, revient à montrer que $\boldsymbol{CLIQUE} \in \boldsymbol{NP}$ puis que ***CLIQUE*** est ***NP-Difficile***

a- Montrons que clique est dans NP

Pour montrer que $\boldsymbol{CLIQUE} \in \boldsymbol{NP}$, pour un graphe donné $\boldsymbol{G} = (\boldsymbol{S}, \boldsymbol{A})$**,** on utilise l'ensemble $\boldsymbol{S'} \subseteq \boldsymbol{S}$ des sommets de la clique comme certificat pour **G**. Vérifier que **S'** est une clique peut être accompli en temps polynomial, en testant si pour toute paire **u, v** ∈ **S'** l'arête **(u, v)** appartient à **A**.

b- Montrons la NP-difficulté de clique

Nous allons montrer que *3-CNF-SAT* se réduit polynomialement à *CLIQUE* c'est-à-dire $\boldsymbol{3\text{-}CNF\text{-}SAT} \leq_P \boldsymbol{CLIQUE}$. L'algorithme de réduction commence avec une instance de $\boldsymbol{3\text{-}CNF\text{-}SAT}$. Soit $\boldsymbol{f} = \boldsymbol{C_1} \wedge \boldsymbol{C_2} \wedge \cdots \wedge \boldsymbol{C_k}$ une formule booléenne sous forme $\boldsymbol{3\text{-}CNF}$ à $\boldsymbol{k}$ clauses. Pour $\boldsymbol{r}$ = $\boldsymbol{1, 2, \ldots k,}$ chaque clause $\boldsymbol{C_r}$ possède exactement trois littéraux distincts $\boldsymbol{l_{r1}}, \boldsymbol{l_{r2}}$ *et* $\boldsymbol{l_{r3}}$.

Nous allons construire un graphe $\boldsymbol{G}$ tel que **f** soit satisfaisable si et seulement si *G* possède une clique de taille $\boldsymbol{k}$. Le graphe $\boldsymbol{G} = (\boldsymbol{S}, \boldsymbol{A})$ est construit de la manière suivante. Pour chaque clause $\boldsymbol{C_r} = (\boldsymbol{l_1} \vee \boldsymbol{l_2} \vee \boldsymbol{l_3})$ de **f**, on place un triplé de sommets $\boldsymbol{v_{r1}}$, $\boldsymbol{v_{r2}}$ et $\boldsymbol{v_{r3}}$ dans $\boldsymbol{S}$. On place une arête entre deux sommets $\boldsymbol{v_{ri}}$ et $\boldsymbol{v_{sj}}$ si les deux conditions suivantes sont satisfaites :

- $\boldsymbol{v_{ri}}$ et $\boldsymbol{v_{sj}}$ sont dans des triplets différents, autrement dit si *r* est diffèrent de *s* ;
- Leurs littéraux correspondants sont ***cohérents***, autrement dit $\boldsymbol{l_{ri}}$ n'est pas la négation de $\boldsymbol{l_{sj}}$.

Ce graphe peut se calculer facilement à partir de f en temps polynomial. A titre d'exemple de cette construction, si l'on se donne

$\boldsymbol{f} = (\boldsymbol{x_1} \vee \neg\boldsymbol{x_2} \vee \neg\boldsymbol{x_3}) \wedge (\neg\boldsymbol{x_1} \vee \boldsymbol{x_2} \vee \boldsymbol{x_3}) \wedge (\boldsymbol{x_1} \vee \boldsymbol{x_2} \vee \boldsymbol{x_3})$, alors *G* est le graphe représenté à la figure suivante

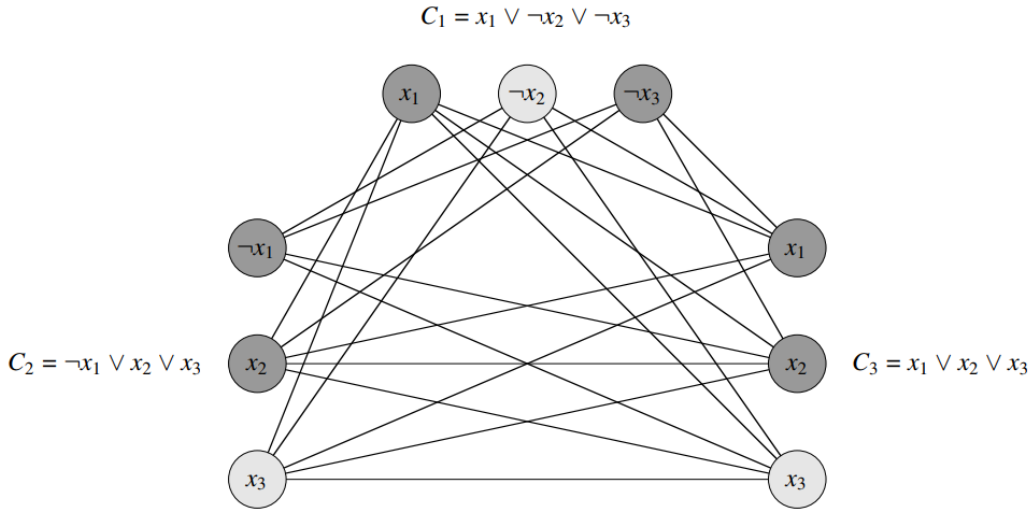

Figure 6 : Graphe issu de la formule $\boldsymbol{f} = (\boldsymbol{x_1} \vee \neg \boldsymbol{x_2} \vee \neg \boldsymbol{x_3}) \wedge (\neg \boldsymbol{x_1} \vee \boldsymbol{x_2} \vee \boldsymbol{x_3}) \wedge (\boldsymbol{x_1} \vee \boldsymbol{x_2} \vee \boldsymbol{x_3})$

Montrons que ce passage de $\boldsymbol{f}$ à ***G est une réduction***.

- D'abord, supposons que $\boldsymbol{f}$ est une assignation satisfaisante. Alors, chaque clause $\boldsymbol{C_r}$ contient au moins un littéral $\boldsymbol{l_{ri}}$ ayant la valeur 1, et chaque littéral de ce type correspond à un sommet $\boldsymbol{v_{ri}}$. En prenant l'un de ces littéraux ayant la valeur vrai dans chaque clause, on obtient un ensemble $\boldsymbol{S}$ de $\boldsymbol{k}$ sommets. Nous affirmons que *S* est une clique. Pour deux sommets quelconques $\boldsymbol{v_{ri}}$, $\boldsymbol{v_{sj}} \boldsymbol{\in S,}$ où $\boldsymbol{r}$ $\boldsymbol{s}$, l'assignation satisfaisante associe la valeur 1 aux deux littéraux $\boldsymbol{l_{ri}}$ et $\boldsymbol{l_{sj}}$ correspondants, et les littéraux ne peuvent donc pas être compléments.

Donc, d'après la construction de *G*, l'arête ($\boldsymbol{v_{ri}}$, $\boldsymbol{v_{sj}}$) appartient à *A*. Le graphe précèdent issu de la formule ***3-CNF*** $\boldsymbol{f} = \boldsymbol{C_1} \wedge \boldsymbol{C_2} \wedge \boldsymbol{C_3}$, où $\boldsymbol{C_1} = (\boldsymbol{x_1} \vee \neg \boldsymbol{x_2} \vee \neg \boldsymbol{x_3})$, $\boldsymbol{C_2} = (\neg \boldsymbol{x_1} \vee \boldsymbol{x_2} \vee \boldsymbol{x_3})$ et $\boldsymbol{C_3} = (\boldsymbol{x_1} \vee \boldsymbol{x_2} \vee \boldsymbol{x_3})$, lors de la réduction de ***3-CNF-SAT*** à ***CLIQUE.*** L'assignation $< \boldsymbol{x1} = 0$ *ou* $1,\ \boldsymbol{x2} = 0,\ \boldsymbol{x3} = 1 >$ est satisfaisante.
Cette assignation vérifie $\boldsymbol{C_1}$ avec $\neg \boldsymbol{x_2}$ et vérifie $\boldsymbol{C_2}$ et $\boldsymbol{C_3}$ avec $\boldsymbol{x_3}$, ce qui correspond à la clique dont les sommets sont en clair.

- Supposons maintenant que **G** contienne une clique de taille k. Comme les littéraux d'une même clause ne sont pas reliés, cette clique contient un littéral exactement dans chaque clause. Montrons alors qu'il existe une assignation qui rend tous ces littéraux vrais. Chaque littéral de cette clique est égal à $\boldsymbol{x_i}$ ou à $\neg \boldsymbol{x_i}$. Pour que ce littéral soit vrai, on impose la valeur 1 ou 0 à la variable correspondante $\boldsymbol{x_i}$. Comme tous les littéraux de la clique sont reliés par une arête, ils ne sont pas contradictoires deux à deux. Ceci signifie que deux littéraux

quelconques de la clique concernent deux variables distinctes x_i et x_j avec $i \neq j$ ou alors ils concernent la même variable x_i mais ils imposent la même valeur à la variable x_i. On obtient alors une assignation cohérente des variables apparaissant dans la clique. En affectant n'importe quelle valeur à chacune des autres variables, on obtient une affectation qui donne la valeur 1 à la formule **f**. On pourrait penser que l'on a montré que ***CLIQUE*** est ***NP- difficile*** uniquement pour les graphes où les sommets vont par trois et où il n'y a pas d'arêtes entre deux sommets d'un même triplet.

En fait, on a montré que ***CLIQUE*** est ***NP-difficile*** uniquement pour ce cas particulier, mais cette preuve suffit pour montrer que ***CLIQUE*** est ***NP-difficile*** pour des graphes quelconques.
D'où le problème de ***la clique est NP-complet***.

v) Couverture

Une ***couverture de sommets*** d'un graphe non orienté $G = (S, A)$ est un sous-ensemble $S' \subseteq S$ tel que si $(u, v) \in A$, alors $u \in S'$ou $v \in S'$(ou les deux). Autrement dit, chaque sommet « couvre » ses arêtes incidentes et une couverture de sommets pour *G* est un ensemble de sommets qui couvre toute les arêtes de *A*. La ***taille*** d'une couverture de sommets est le nombre de sommets qui la composent. Par exemple, le graphe de la figure suivante a une couverture de sommets $\{w, z\}$ de taille 2.

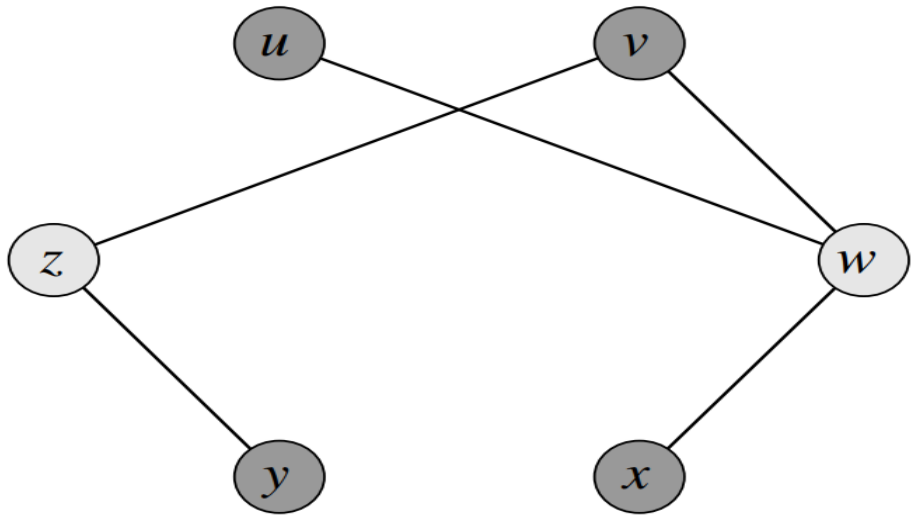

Figure7 : Exemple de couverture de sommets 2

Le problème de la couverture de sommets consiste à trouver une couverture de

sommet de taille minimale dans un graphe donné. ***COUVERTURE-SOMMETS*** est un problème d'optimisation problème d'optimisation comme un problème de décision, on souhaite déterminer si un graphe possède une couverture de sommets d'une taille *k* donnée. Si on lui donne l'aspect formel d'un langage, on aura

COUVERTURE-SOMMETS = ***{<G, k >: le graphe G possède une couverture de sommets de taille k}***.

Pour montrer que ***COUVERTURE-SOMMETS*** est ***NP-complet*** commençons par montrer que ***COUVERTURE-SOMMETS*** ∈ NP.

- Supposons qu'on ait un graphe ***G***= ***(S, A)*** et un entier ***k***. Le certificat que nous choisissons est la couverture de sommets $S \subseteq S$ elle-même. L'algorithme de vérification affirme que |***S***| = ***k***, puis il vérifie, pour chaque arête $(u, v) \in A$, que $u \in S$ ou $v \in S$. Cette vérification peut être effectuée en temps polynomial.

- On démontre ensuite que le problème de la couverture de sommets est ***NP-difficile*** en montrant que ***CLIQUE*** $\leq_P$ ***COUVERTURE-SOMMETS***. Cette réduction est fondée sur la notion de graphe complémentaire. Étant donné un graphe non orienté ***G*** = ***(S, A),*** on définit le ***complémentaire*** de ***G*** comme ***G***'= ***(S, $\bar{A}$)*** où $\bar{A}$= ***{(u, v) : u, v ∈ S, u ≠ v, et (u, v) ∉A}.***

Autrement dit, ***G*** est le graphe contenant exactement les arêtes qui ne sont pas dans ***G***. La figure suivante montre un graphe et son complémentaire, et illustre la réduction de ***CLIQUE*** à ***COUVERTURE-SOMMETS***.

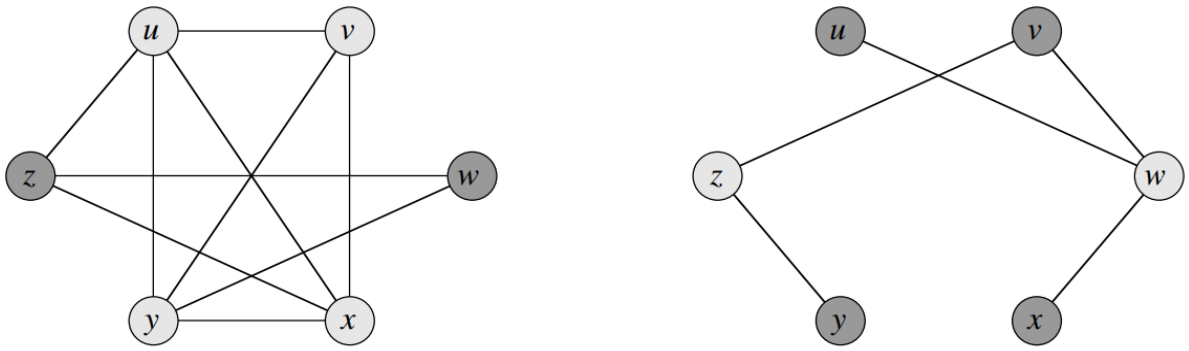

Figure 8 : Réduction de clique à couverture de sommets

L'algorithme de réduction prend en entrée |***S***|- ***k,*** une instance <***G, k*** > du problème de la clique. Il calcule le complémentaire ***G'***, ce qui se fait facilement en temps polynomial. Le résultat de l'algorithme de réduction est l'instance <***G,*** |***S***|-***k***> du problème de la couverture de sommets.

Pour compléter la démonstration, on montre que cette transformation est bien une réduction : le graphe *G* possède une clique de taille ***k*** si et seulement si le graphe ***G'*** possède une couverture de sommets de taille $|S| - k$. Supposons que ***G*** ait une clique $S' \subseteq S$ avec $|S'| = k$. Nous affirmons que ***S-S'*** est une couverture de sommets de ***G'***. Soit ***(u, v)*** une arête quelconque de $\bar{A}$. Alors, $(u, v) \notin A$, ce qui implique qu'au moins un des sommets ***u*** et ***v*** n'appartient pas à ***S'***, puisque toute paire de sommets de ***S*** est reliée par une arête de ***A***. De manière équivalente, l'un au moins des sommets *u* ou *v* se trouve dans $S - S'$, ce qui signifie que l'arête ***(u, v)*** est couverte par $S - S'$. Comme ***(u, v)*** a été choisi arbitrairement dans $\bar{A}$, toute arête de $\bar{A}$ est couverte par un sommet de $S - S'$. Donc, l'ensemble $S - S'$, qui a une taille forme une couverture de sommets pour ***G'***.

Réciproquement, supposons que ***G*** ait une couverture de sommets $S' \subseteq S$, où $|S'| = |S| - k$. Alors, quels que soient $u, v \in S$, si $(u, v) \in \bar{A}$, on a $u \in S'$ ou $v \in S'$ ou les deux. La contraposée de cette implication est : quels que soient $u, v \in S$, si $u \notin S'$ et $v \notin S'$, alors $(u, v) \in A$. Autrement dit, $S - S$ est une clique et a une taille $|S| - |S'| = k$.

V.CONSEQUENCE SI P=NP

- Parmi ces problèmes **NP**-Complets, se trouve la cryptanalyse des systèmes de cryptographie à clé publique, utilisée notamment pour la sécurisation des transactions bancaires, et dont la sécurité repose sur l'assertion qu'il n'existe pas d'algorithme polynomial pour casser le système cryptographique. En d'autres termes, la sécurité des transactions bancaires repose sur la supposition que $\mathbf{P} \neq \mathbf{NP}$. S'il s'avérait que P=NP, alors il suffirait juste de trouver un algorithme polynomial pour casser tous les systèmes de sécurité basés sur la clés publique .

- Les problèmes **NP**-Complets concernent un grand nombre de domaines différents : la biologie, avec par exemple l'algorithme de détermination de la séquence d'ADN qui correspond le mieux à un ensemble de fragments, ou le calcul de solution optimales en économie (équilibres de Nash), ou dans les processus de fabrication ou de transport.

- La factorisation des grands nombres, donc craquer les chiffrements RSA qui securisent pas mal de communications et transactions sur internet

- Résoudre en temps polynomial de repliement de protéines , ce qui permettrait peut être de trouver des molécules pour soigner un tas de maladies comme le cancer ; si P=NP cela peut être vraiment révolutionnaire dans la médecine
- En trouvant un algorithme qui peut prouver que P=NP, nous permettrait aussi de gagner 1 million de dollars…

VI.APPLICATIONS DE LA NP-COMPLETUDE

Il est possible de réduire (traduire) certains problèmes d'intelligence artificielle au problème SAT afin de résoudre efficacement ces problèmes.

▪ Diagnostic

Le diagnostic de systèmes statiques consiste à déterminer si un système a un comportement défectueux étant donnée l'observation des entrées et sorties du système. Le modèle du système peut être traduit en un ensemble de contraintes (disjonctions) : pour chaque composant du système, une variable est créée qui est évaluée à *vraie* si le composant a un comportement anormal (***Ab****normal*). Les observations peuvent être également traduites par un ensemble de disjonctions. L'assignation trouvée par l'algorithme de satisfaisabilité est un diagnostic.

▪ Planification classique

Le problème de planification classique consiste à trouver une séquence d'actions menant d'un état du système à un ensemble d'états. Étant donnée une longueur maximale du plan et un ensemble de variables d'état booléennes permettant de décrire l'état du système, on crée les

variables propositionelles pour tout et toute variable . La variable est *vraie* si la variable d'état est *vraie* après l'action numéro . On crée également les variables pour tout et toute action . La variable est *vraie* si l'action numéro est . Il est alors possible de transformer le modèle du système en un ensemble de clauses. Par exemple, si l'action fait passer la variable à *vrai* lorsque celle-ci est *fausse*, alors la CNF contiendra une clause (ce

qui est traduit par la clause). L'assignation trouvée par l'algorithme de satisfaisabilité peut être immédiatement traduite en plan.

La planification classique par SAT est très efficace si on connaît la longueur *n* du plan[réf. nécessaire]. Si cette valeur n'est pas connue, on peut chercher des plans pour une valeur incrémentale, ce qui est parfois coûteux (notamment parce que la CNF est non satisfaisable jusqu'à $n - 1$).

- **Model checking**

Une approche semblable a été utilisée pour le *model checking* (vérification de propriétés d'un modèle). La principale différence est que le *model checking* s'applique à des trajectoires de longueur infinie contrairement à la planification. Cependant, si l'espace d'états du système est fini, toute trajectoire infinie boucle à un certain point et on peut borner la taille des trajectoires qu'il est nécessaire de vérifier. Le *bounded model checking* tire parti de cette propriété pour transformer le problème de *model checking* en un certain nombre de problèmes de satisfaisabilité.

- **Cryptographie**

La complexité du problème SAT est une composante essentielle de la sécurité de tout système de cryptographie.

Par exemple, une fonction de hachage sécurisée constitue une boîte noire pouvant être formulée en un temps et un espace fini sous la forme d'une conjonction de clauses normales, dont les variables booléennes correspondent directement aux valeurs possibles des données d'entrée de la fonction de hachage, et chacun des bits de résultat devra répondre à un test booléen d'égalité avec les bits de données d'un bloc de données d'entrées quelconque. Les fonctions de hachages sécurisées servent notamment dans des systèmes d'authentification (connaissance de données secrètes d'entrée ayant servi à produire la valeur de hachage) et de signature (contre toute altération ou falsification « facile » des données d'entrée, qui sont connues en même temps que la fonction de hachage elle-même et de son résultat).

La sécurité de la fonction de hachage dépend de la possibilité de retrouver un bloc de données d'entrée arbitraire (éventuellement différent du bloc secret de données pour lequel une valeur de hachage donnée a été obtenue) permettant d'égaliser la valeur binaire retournée par la fonction de hachage. Une méthode d'exploration systématique de toutes les valeurs possibles des données d'entrée et pour lesquelles on applique la fonction de hachage comme un oracle, permet effectivement de satisfaire à la question de une valeur de hachage égale à celle cherchée, mais sa complexité algorithmique sera exponentielle (en fonction de la taille maximale en bits des données d'entrée de la fonction de hachage).

Chercher à égaliser une valeur de hachage avec des variables d'entrée de valeur inconnue constitue un problème SAT (et comme en pratique les données d'entrée de la fonction de hachage sont constituées de nombreux bits, ce sera un problème *n*-SAT avec *n* assez élevé (et en tout cas supérieur ou égal à 3). On sait alors que ce problème est réductible en temps polynomial à un problème 3–SAT, qui est NP-complet.

La sécurité de la fonction de hachage sera fortement liée au fait de l'impossibilité de la réduire plus simplement (que par un algorithme NP-complet de complexité suffisante) en réduisant la forme conjonctive des clauses qui la définissent simplement, pour y isoler des sous-ensembles de variables d'entrées de taille plus restreinte, mais suffisante pour déterminer une partie du résultat de la fonction de hachage dont l'ordre *n* sera alors très inférieur, et pour ne plus avoir à explorer que les valeurs possibles de ces seuls sous-ensembles de variables pour satisfaire la condition d'égalité du résultat, sans avoir à tester toutes les autres variables dont la valeur peut

être fixée arbitrairement à certaines valeurs déterminées par un algorithme plus simple (apprentissage de clauses, approches prospectives ci-dessus).

- **Logique modale**

Le problème SAT peut être utilisé comme sous-routine pour résoudre le problème de satisfiabilité de la logique modale

CONCLUSION

La perfection ne fait probablement pas partie de ce monde et même si elle y était, nos ordinateurs (ainsi que nous-mêmes) seraient encore très loin de l'atteindre. Il n'existe donc pas de technique exacte ou d'algorithme universel : à titre d'exemple, seul le filtrage par consistances n'est pas suffisant pour aboutir à une solution quasi optimale pendant un temps limité, ou encore, la simple application de quelque heuristique (comme le *best first* ou le *first failled*) n'est pas la réponse définitive. Une règle d'informaticiens qui est toujours valide est *''chercher si une solution existe déjà avant de se pencher sur le problème''* (Emmanuel Kounalis). L'application du meilleur algorithme peut être parfois très coûteuse si l'on a "*gaspillé*" énormément de temps pour le trouver, de plus que l'on pourrait remarquer par la suite qu'il n'était pas le meilleur.

Il faut considérer chaque situation, le type et la quantité des donnés, les moyens et le temps à notre disposition pour choisir les techniques et les heuristiques à appliquer. Souvent le monde de l'industrie préfère une réponse approximative mais rapide et efficace plutôt que l'utopique réponse parfaite

BIBLIOGRAPHIE

- Introduction à l'algorithmique, Cours et Exercices 2ème Edition, DUNOD
- Anciens exposés sur la NP-complétude de nos aînés académique

WEBOGRAPHIE

- https://fr.m.wikipedia.org/wiki/Probl%C3%A8me_NP-complet#:~:text=La%20NP%2Dcompl%C3%A9tude%20ne%20donne,une%20partie%20d'entre%20elles.
- https://fr.slideshare.net/mobile/sanaaroussi3/chapitre-3-npcompltude
- https://24pm.com/117-definitions/446-np-complet

FICHE DE TRAVAUX DIRIGES

Exercice 1 :

Montrer que la classe P, vue en tant qu'ensemble de langages, est fermée pour l'union, l'intersection, la concaténation, le complément et l'opération Kleene étoile. Autrement dit, si L1, L2 ∈ P, alors L1 ∪ L2 ∈ P,

Exercice 2 : Réduction polynomiale

.

i- Propriétés sur l'ordre ≤

Soient A et B deux problèmes de décision.

Une réduction de A vers B est une fonction f : IA → IB calculable en temps polynomial telle que w ∈ Oui(A) si et seulement si f(w) ∈ Oui(B). Nous notons A ≤ B lorsque A se réduit à B.

a) Montrer que ≤ est réflexive (L ≤ L).

b) Montrer que ≤ est transitive (c'est-à-dire L1 ≤ L2, L2 ≤ L3 impliquent L1 ≤ L3.)

c) Montrer que P = NP si et seulement si 3-SAT ∈ P.

ii- Réduction

Soient A, B et Q des problèmes de décision. Supposons que A est dans P, et que B est NP-dur. Dire si les affirmations suivantes sont vraies ou fausses :

1. Si A se réduit polynômialement à Q, alors Q est dans P.
2. Si Q se réduit polynômialement à A, alors Q est dans P.
3. Si Q se réduit polynômialement à B, alors Q est NP −dur.
4. Si B se réduit polynômialement à Q, alors Q est NP –dur

Exercice 3 : Le problème 2-SAT est dans P

Nous allons considérer de cet exercice des fonctions booléennes, c'est-à-dire de fonctions de f de $\{1,0\}n \rightarrow \{1,0\}$.

- Les variables ne peuvent prendre que deux valeurs, vrai (codé par 1) ou faux (codé par 0).
- La fonction booléenne f est composée de variables et d'opérateurs comme négation (¬) la conjonction (∧) la disjonction (∨), l'implication →.

Voici la table de vérité pour les opérateurs booléens cités ci-dessus :

u	v	$\neg u$	$(u \wedge v)$	$u \vee v$	$u \rightarrow v$
0	0	1	0	0	1
0	1	1	0	1	1
1	0	0	0	1	0
1	1	0	1	1	1

Considérons une fonction t : $U \rightarrow \{0,1\}$. On dira que la fonction t satisfait la fonction f si la fonction f retourne 1 avec les valeurs de t en entrée.

1. Nous allons considérer la fonction φ_1 : telle que $\{1,0\}^3 \rightarrow \{1,0\}$.

$$\varphi_1(x,y,z) = (y \vee \neg z) \wedge (\neg y \vee z) \wedge (y \vee \neg x).$$

Donner une fonction t : U → {0,1} telle que t satisfait φ_1.

2. Que se passe-t-il pour φ_2

$$\varphi_2(x,y,z) = (y \vee z) \wedge (\neg y \vee z) \wedge (\neg z \vee x) \wedge (\neg z \vee \neg x) \ ?$$

Une clause est une fonction de f de $\{1,0\}^n \rightarrow \{1,0\}$ composée de variables et d'opérateurs comme négation (¬) et la disjonction (∨).

Par exemple $C(u_2, u_3) = (u_2 \vee \neg u_3)$ est une clause.

Donnons la définition du problème 2-SAT

Données : un ensemble U de variables $\{u_1, u_2, \ldots, u_n\}$ et une formule logique $\varphi = C_1 \wedge \cdots \wedge C_l$ des clauses de 2 littéraux.

Existe-t-il une fonction t : $U \rightarrow \{0,1\}$ telle que t satisfait φ ?.

3. Montrer que $u \vee v = (\neg u \Rightarrow v) \wedge (\neg v \Rightarrow u)$

   A partir d'une instance (U, φ) de 2-SAT, nous construisons un graphe orienté $G_\varphi = (V, E)$ tel que

   - un sommet par un littéral de U.
   - un arc par implication (en transformant chaque clause par deux implications)

4. Dessiner les graphes $G_{\varphi 1}$ et $G_{\varphi 2}$
5. Montrer que si G_φ possède un chemin entre u et v, alors il poss`ede aussi un chemin entre $\neg v \ et \ \neg u$.
6. Montrer qu'il existe une variable u de U tel que $G_\varphi$contient un cycle entre u vers $\neg u$ si et seulement si φ n'est pas satisfiable.

Exercice 4 : Différentes Variantes du problème cycle hamiltonien

Nous allons supposer que le problème Cycle Hamiltonien est NP-Complet.

Cycle Hamiltonien

Données : un graphe non-orienté G.

Question : G contient-il un cycle hamiltonien?

a) Considérons le problème Chaine Hamiltonien suivant :

Chaine Hamiltonienne

Données : un graphe non-orienté G, deux sommets u et v distincts de G.

Question : G contient-il une chaine hamiltonienne entre u et v?

Montrer que le problème Chaine Hamiltonien est NP-complet

b) Le problème Chaine est le problème de décision suivant

Chaine

Données : un graphe non-orienté G de n sommets, deux sommets u et v distincts de G.

Question : G contient-il une chaine de longueur n/2 entre u et v?

Montrer que le problème Chaine est NP-complet

c) Chevaliers de la table ronde

Etant données n chevaliers, et connaissant toutes les paires de féroces ennemis parmi eux, est-il possible de les placer autour d'une table circulaire de telle sorte qu'aucune paire de féroces ennemis ne soit côte à côte?

d) Considérez le problème Voyageur de Commerce :

Données : Un graphe complet G muni d'une fonction d de coût positif sur les arrêtes, et un entier k.

Question : Existe-t-il un circuit passant au moins une fois par chaque ville et dont la somme des coûts des arrêtes est au plus k?

- Montrer que Voyageur de Commerce est NP-complet même si la fonction poids respecte l'inégalité triangulaire. (C'est-`a-dire que si les arrêtes vérifient l'inégalité triangulaire, c'est-à-dire si pour tout triplet de sommets $(u, v, w), on\ a\ d(u, v) \leq d(u, w) + d(w, v).)$
- Montrer que le problème du voyageur de commerce ayant une métrique respectant l'inégalité triangulaire est dans NP-complet.

Exercice 5 :

Montrer que le problème du chemin hamiltonien peut être résolu en temps polynomial sur les graphes orientés sans circuits. Donner un algorithme efficace pour le problème.

Exercice 6 : Problème de clique maximum

Considérons le problème de décision ***CLIQUE***: **Données** : un graphe non-orienté $\boldsymbol{G = (V, E)}$ et un entier $\boldsymbol{k}$. **Question** : Existe-t-il une clique de taille *k*?

1. Nous noterons par G^c le complémentaire du graphe **G**. Montrer que **G** a une clique de taille k si et seulement si G^ca une couverture de sommets

   de taille $\boldsymbol{n - k}$.

2. Montrer que le problème **CLIQUE** est NP-complet.

Nous allons travailler sur une restriction du problème **CLIQUE** en considérant uniquement les graphes dans lesquels tous leurs sommets sont de degré au plus 3. Nous le noterons ***3-CLIQUE***.

3. Montrer que ***3-CLIQUE*** est dans **NP**

4. Trouver l'erreur dans le raisonnement suivant :

Nous savons que le problème **CLIQUE** est NP-complet, il suffit donc de présenter une réduction de ***3-CLIQUE*** à **CLIQUE.** Etant donné un graphe G' dont les sommets sont de degré inférieur à 3, et un entier **k**, la réduction laisse inchangée le graphe et le paramètre **k** : clairement le résultat de la réduction est une entrée possible pour le problème CLIQUE. Par ailleurs, la réponse aux deux problèmes est identique. Cela prouve la justesse de la réduction et, par conséquent, la NP-complétude de la **3-CLIQUE**.

5. Donner un algorithme polynomial pour le problème ***3-CLIQUE***