Chapitre 11

Classes de complexité

I Classes de complexité pour un problème de décision

I.1 Complexité

Définition 11.1

Soit $\mathcal{P}$ un problème abstrait défini par une fonction $f : I \to S$. On considère un algorithme $\mathcal{A}$ "abstrait" qui le résout.
On y associe un problème *concret* en imposant des choix sur *l'encodage* des entrées et des sorties.
La *complexité du problème concret* correspond à la complexité de l'implémentation de l'algorithme $\mathcal{A}$ (sa réalisation concrète) pour le choix d'encodage.
C'est-à-dire un ordre de grandeur du nombre d'opérations élémentaires réalisées par l'implémentation de l'algorithme pour une entrée en fonction de la *taille de l'entrée* (espace occupé en mémoire pour son encodage).

Remarque 11.2 - *cas du graphe*

La taille de l'encodage pour un graphe :

- matrice d'adjacence : $\mathcal{O}(|S|^2)$
- liste d'adjacence : $\mathcal{O}(|S| + |A|)$

Exemple 11.3 - *taille de l'endage des entiers*

Un entier n s'encode en $\mathcal{O}(\log_2(n))$, taille de l'écriture binaire.
Une complexité en $\mathcal{O}(n)$ où n est l'entier en entrée, est donc exponentielle pour un algorithme dont l'entrée est n, est $\mathcal{O}(n) = \mathcal{O}(2^{\log_2(n)})$. C'est donc exponentiel en la taille de l'entrée !

Remarque 11.4 - *relative aux problèmes*

Selon l'importance des entiers dans le problème, on peut "ignorer" que les entier prennent beaucoup de place :

- pour un graphe pondéré, la taille des entiers est ignorée.
- pour le problème du sac à dos, la taille des entiers est prise en compte.

Exemple 11.5

- **Instance** : un entier n
- **Question** : $v_2(n) \geq v_3(n)$? (en parlant de valuation 2-adique)

```
1  Algo(n):
2      tmp = n
3      tant que tmp%2 == 0 ou tmp%3==0:
4          si tmp%6==0:
5              tmp = tmp/6
6          sinon si tmp%2==0:
7              renvoyer Vrai
8          sinon si tmp%3==0:
9              renvoyer Faux
10     renvoyer Vrai // v2(n) = v3(n)
```

La complexité temporelle est en $\mathcal{O}(\log_6(n)) = \mathcal{O}(\log_2(n)) = \mathcal{O}(|n|)$. C'est une complexité linéaire en la taille de l'entrée : $|n| = \log_2(n)$.

Moralité : toujours se ramener à une complexité en fonction de la taille de l'entrée.

Remarque 11.6

La taille de l'entrée est à rapprocher de la taille du mot de $\Sigma = \{0; 1\}$ que l'on écrit initialement sur une machine de Turing exécutant l'algorithme sur ce mot.

I.2 Classe P

Définition 11.7 - *classe P*

On appelle *classe P* (ou *classe PTIME*) l'ensemble de *problèmes de décision* que l'on peut résoudre à l'aide d'un algorithme de complexité polynomiale en la taille de l'entrée (pour un choix d'encodage).

Remarque 11.8

Généralement, on donne la complexité sous la forme $\mathcal{O}(|e|^k)$ avec k un entier (degré de l'expression polynomiale de la complexité) et $|e|$ désignant la taille de l'entier.

Remarque 11.9

Les problèmes de la classe P sont raisonnablement traitables. (Attention, on parle toujours uniquement de problème de décision).
Pour considérer des problèmes d'optimisation, on se ramène à des problèmes de décision en introduisant un seuil à dépasser pour la fonction de coût.

Remarque 11.10 - *classe EXP*

La *classe EXP* (ou *classe EXPTIME*) est la classe de problèmes de décision pour lesquels il existe un algorithme de résolution de complexité exponentielle :

$$\mathcal{O}\Big(P(|e|)\exp(|e|)\Big)$$

où P est un polynôme réel et $|e|$ est la taille de l'entrée.

Remarque 11.11 - *comparaison entre EXP et P*

On a P $\subset$ EXP car $e^n \gg P(n)$ pour $n \to +\infty$.

Remarque 11.12

La complexité, soit le nombre d'opérations élémentaires, dépend du modèle de calcul choisi. Il en existe plusieurs.
En MPI, on se place dans le cadre d'un modèle de calcul intuitif, qui correspond au fonctionnement d'un ordinateur exécutant un programmation en C ou en OCaml. Les opérations élémentaires sont alors les suivantes :

- écriture
- lecture
- opération arithmétique
- etc (cf MP2I)

De plus on fait l'hypothèse que la machine dispose d'une *mémoire infinie* :

- la pile d'appels ne déborde jamais
- il n'y a pas de limite à la quantité de données allouable sur la pile, le tas ou bien le segment de données.

Un autre modèle de calcul classique est celui des machines de Turing : on compte le nombre de transitions empruntées au cours d'un calcul.

Exemple 11.13 - *problème du dernier exemple*

Le problème précédent (valuations) est de classe P car linéaire et particulier polynomial.

Exemple 11.14 - *problème* ***2-SAT***

- **Instance** : φ une formule de la logique propositionnelle sous forme normale conjonctive dont les clauses sont de taille au plus 2.
- **Question** : φ est-elle satisfiable ?

On a **2-SAT** $\in$ P. En effet, on peut construire le graphe d'implications puis lui appliquer l'algorithme de Kosaraju qui renvoie les composantes fortement connexes du graphe d'implications. On vérifie que pour $x \in \mathcal{V}_\varphi$, x et $\neg x$ ne sont pas dans la même composante fortement connexe. $\varphi = C_1 \wedge \cdots \wedge C_k$ de taille $\mathcal{O}(k) = |\varphi|$.

Construction du graphe :

- au plus $4k$ sommets (un par littéral)
- au plus $2k$ arêtes (deux par clause)

se fait en $\mathcal{O}(k)$
Kosaraju : $\mathcal{O}(|\mathcal{A}| + |\mathcal{S}|)$ (deux parcours), se fait en
à finir

I.3 Classe NP

Définition 11.15

Pour un problème de décision, on appelle *certificat* une donnée que l'on peut ajouter à l'entrée permettant de vérifier que l'entrée est une instance positive.

Exemple 11.16

Pour le problème **SAT**, la donnée d'une valuation est un certificat. En effet, on vérifie que la valuation satisfait la formule pour vérifier qu'elle est satisfiable donc que c'est une instance positive. Pour toute formule φ, il existe une valuation v telle que l'algorithme vérificateur confirme que φ est satisfiable est satisfable.

Exemple 11.17 - *lié au problème K-COLOR*

- **Instance** : $G = (S, A)$ un graphe, $k \in \mathbb{N}$.
- **Question** : sur l'existence de $c : S \to [\![1, k]\!]$ un coloriage des sommets à k couleurs tel que :

$$\forall (u, v) \in A,\ c(u) \neq c(v)$$

On peut chosiir la donnée d'une fonction $c : S \to [\![1, k]\!]$ comme certificat.

Définition 11.18 - *classe NP*

La classe NP est l'ensemble des problèmes de décision pour lesquels il existe un *algorithme vérificateur* permettant de *vérifier* qu'une instance est positive à partir d'un couple (instance, certificat) en temps polynomial en la taille de l'instance, avec un certificat de taille polynomiale en fonction de cette taille.

Exemple 11.19 - *problème* ***SAT***

Soit φ une instance positive (une formule satisfiable) et v un certificat associé à φ (un modèle de φ) :

$$|v| = |\mathcal{V}_\varphi| = |\varphi|$$

L'algorithme vérificateur repose sur un parcours de l'arbre représentant φ en $\mathcal{O}\Big(|\varphi|\Big)$

Remarque 11.20

L'algorithme vérificateur prend en entrée :

- un encodage de l'instance
- un encodage du certificat

deux chaînes de caractères ! Si on choisit un certificat booléen, pour un instance i^+ positive, on note `verif` l'algorithme tel que :

- `verif(`i^+`,true)` renvoie `true`
- `verif(`i^+`,false)` renvoie un booléen arbitraire

et pour i^- une instance négative :

- `verif(`i^-`,true)` renvoie `false`
- `verif(`i^-`,false)` renvoie `false`

Si on a un tel algorithme, alors on sait résoudre le problème avec cet algorithme et le certificat n'est pas nécessaire. Donc on ne pas prendre directement la solution comme un certificat ? A montrer à ChatGPT, il saura m'expliquer.

Remarque 11.21 - *sur la définition*

Pour tout instance positive, l'algorithme vérificateur doit renvoyer Vrai pour l'instance accompagnée d'un certificat valide en temps polynomial en la taille de l'entrée.

Pour toute instance négative, l'algorithme vérificateur doit renvoyer Faux pour cette instance, quel que soit le certificat qui l'accompagne.

Définition 11.22 - *formelle de classe NP*

Soit $\mathcal{P}$ un problème de décision défini par sa fonction de prédicat $f : I \to \{Vrai, Faux\}$. $\mathcal{P}$ est de classe NP lorsqu'il existe :

1. Σ un alphabet permettant d'encoder un ensemble C de certificats
2. $\mathcal{P}'$ un problème de prédicat :
$$f' : I \times \Sigma^* \to S = \{Vrai, Faux\}$$
3. $g : I_1 \to I_2$ une fonction polynomiale telle que pour tout $e \in I$, $f(e)$ est Vrai ssi :
$$\exists c \in C, |c| \leq g(\mathtt{e}) \quad \text{et} \quad f'(e, |c|) = Vrai$$
($|\cdot|$ désigne la taille de représentation)
4. le problème $\mathcal{P}'$ est dans $\mathcal{P}$.

L'algorithme en temps polynomial qui résout $\mathcal{P}'$ est appelé *algorithme vérificateur*.

Remarque 11.23 - *sur la définition*

Dans la 3. : $|c| \leq g(|e|)$ signifie que le certificat est de taille polynomiale en fonction de la taille de l'instance.

Remarque 11.24

Pour montrer en pratique qu'un problème est dans NP :

- on exhibe un certificat de taille polynomiale en fonction de l'instance
- on donne un algorithme vérificateur de complexité polynomiale en fonction de l'instance.

Exemple 11.25

On considère le problème CYCLE HAMILTONIEN :

- **Instance** : $G = (S, A)$ un graphe
- **Question** : Existe-t-il un cycle passant exactement par tous les sommets de S ?
- **Certificat** : une permutation σ de S.

```
1 | VERIFICATEUR(G,σ):
2 |     n = |S|
3 |     vérifier que σ est dans S_n.
4 |     pour i=1 jusqu'à n-1:
5 |         Si (σ(i),σ(i+1)) ∉ A:
6 |             renvoyer false
7 |     renvoyer ((σ(n),σ(1)) ∈ A)
```

Cet algorithme est de complexité en $\mathcal{O}(|\mathcal{S}|)$ donc en temps polynomial en fonction de la taille de G en mémoire : $|S| + |A|$ et $|\sigma| = |S|$ également polynomiale.

Proposition 11.26 - *classe P implique classe NP*

P $\subset$ NP.

Démonstration

On peut prendre n'importe quel certificat, on utilise l'algorithme résolvant $\mathcal{P}$ en temps polynomial comme algorithme vérificateur.

Remarque 11.27 - *sens réciproque*

Le sens réciproque est un problème ouvert. On ne connaît aucun exemple de problème de classe NP pour lequel on a une démonstration du fait qu'il n'est pas de classe P.

I.4 Le problème d'optimisation

Pour étudier la classe de complexité d'un problème d'opitimisation , one le transforme en problème de décision en ajoutant un *seuil* à l'entrée. On répond alors à la question :
"Existe-t-il une solution réalisable dont la valeur atteinte par la fonction de coût est supérieur (maximisation) ou inférieur (minimisation) à un seuil s ?"

Le problème de décision associé est "moins compliqué". En particulier, si pour le problème de décision, on ne sait pas trouver d'algorithme en temps polynomial, on ne saura pas non plus pour le problème d'optimisation : il est

légitime de s'intéresse à des *algorithmes d'approximation*.

Remarque 11.28

Si le problème de décision est de classe P, on peut s'approcher le plus possible de la valeur optimale par dichotomie en utilisant plusieurs appels à l'algorithme de résolution en temps polynomial.

On introduit la notation hors programme suivante.
La classe des problèmes d'optimisation dont le problème de décision associé est de classe P s'appelle la classe PO.
De même, pour la classe NP, la classe NPO.

II Réduction polynomiale

Définition 11.29 - *réduction polynomiale*

Soit deux problèmes de décision $\mathcal{P}_1$ et $\mathcal{P}_2$ définis par f_1 et f_2 leurs fonctions de prédicats associées définies sur un I_1 et I_2 respectivement. On dit que *$\mathcal{P}_1$ se réduit à $\mathcal{P}_2$ de façon polynomiale* (noté $\mathcal{P}_1 \leq_m^{\mathrm{P}} \mathcal{P}_2$) lorsqu'il existe $g : I_1 \to I_2$ calculable avec un algorithme de complexité polynomiale telle que :

$$\forall e \in I_1,\ f_1(e) = f_2\Big(g(e)\Big)$$

Remarque 11.30

On a la même définition de réduction many-to-one que lorsqu'on discute de décidabilité, on uniquement l'hypothèse supplémentaire de l'existence d'un algorithme polynomial qui transforme les instances de $\mathcal{P}_1$ en des instances de $\mathcal{P}_2$.

Proposition 11.31 - *$\leq_m^{\mathrm{P}}$ est presque une relation d'ordre*

La relation $\leq_m^{\mathrm{P}}$ est réflexive, transitive, mais non antisymétrique.

Voir démo 1

Proposition 11.32

Soit $\mathcal{P}_1$ et $\mathcal{P}_2$ deux problèmes de décision. Si $\mathcal{P}_1 \leq_m^{\mathrm{P}} \mathcal{P}_2$ et $\mathcal{P}_2$ est de classe P, alors $\mathcal{P}_1$ est de classe P.

Démonstration

Il suffit de combiner l'algorithme calculant g et l'algorithme qui résout $\mathcal{P}_2$ en temps polynomial.

Voir suite par Gwénolé.

III NP-completude, NP-difficile

III.1 Définitions et problème SAT

Définition 11.33

Un problème de décision est *NP-difficile* ou *NP-dur* lorsqu'il est possible de se ramener à tout problème de classe NP par une réduction polynomiale depuis ce problème.
Moralement, il est plus difficile que tous les problèmes NP, mais on peut s'y ramener.

Définition 11.34

Soit $\mathcal{P}$ un problème de décision. $\mathcal{P}$ est NP-difficile si et seulement si :

$$\forall \mathcal{P}' \in \text{NP},\ \mathcal{P}' \leq_m^{\text{P}} \mathcal{P}$$

Si de plus $\mathcal{P}$ est dans NP, alors $\mathcal{P}$ est *NP-complet*.

Remarque 11.35

Pour montrer qu'un problème est NP-diffficile, on montre que **SAT** se réduit polynomialement à ce problème : **SAT** $\leq_m^{\text{P}} \mathcal{P}$. On utilise donc la transitivité de $\leq_m^{\text{P}}$.

III.2 Rédution polynomiale et NP-difficulté

Proposition 11.36

Soit $\mathcal{P}_1$ et $\mathcal{P}_2$ deux prolbèmes de décision. Si $\mathcal{P}_1 \leq_m^{\text{P}} \mathcal{P}_2$ et $\mathcal{P}_1$ est NP-difficile, alors $\mathcal{P}_2$ est NP-difficile.

Démonstration

$\leq_m^{\text{P}}$ est une relation transitive. On va l'utiliser. On a :

$$\begin{cases} \forall \mathcal{P} \in NP,\ \mathcal{P} \leq_m^{\text{P}} \mathcal{P}_1 \\ \mathcal{P}_1 \leq_m^{\text{P}} \mathcal{P}_2 \end{cases}$$

Par transitivité,

$$\forall \mathcal{P} \in NP,\ \mathcal{P} \leq_m^{\text{P}} \mathcal{P}_2$$

$\mathcal{P}_2$ est bien NP-difficile.

Remarque 11.37

Pour montrer qu'un problème $\mathcal{P}$ est NP-complet :

- on montre qu'il est de classe NP l'aide d'un certificat et d'un algorithme vérificateur.
- on montre qu'il est NP-difficile en choisissant un problème $\mathcal{P}_{\text{ref}}$ NP-difficile de référence (**SAT** , **CNF-SAT** ou **3-SAT** sont permis), et on montre que $\mathcal{P}_{\text{ref}} \leq_m^{\text{P}} \mathcal{P}$.

III.3 Exemple du problème FNC-SAT ou CNF-SAT

Définition 11.38 - *problème* ***CNF-SAT***

- **Instance** : φ une formule de la logique propositionnelle sous forme normale conjonctive (FNC)
- **Question** : Est-ce que φ est satisfiable ?

Proposition 11.39

CNF est NP-complet.

Remarque 11.40

Utilisable sans rappel au concours. (on l'a dit avant, c'est le cas pour trois problèmes)

Démonstration

Par étapes habituelles. La méthode suivie est décrite en *italique*. Montrons que **CNF-SAT** est de classe NP.
On choisit comme ensemble de certificats l'ensemble des valuations. *Justifions que pour une instance positive au problème, il existe un certificat.* Pour une formule φ satisfiable, il existe v_φ un modèle de φ sur $\mathcal{V}_\varphi$ l'ensemble des variables que φ comporte. v_φ est de taille en $\mathcal{O}(|\varphi|)$.

Un algorithme vérificateur : on parcourt l'arbre de la formule en utilisant la valuation pour obtenir les valeurs de vérité pour les variables et on procède récursivement. Moralement, on regarde si la valuation est un modèle.

```
1   type formule =  X of int | Not of formule | Or of formule*formule
2   | And of formule*formule | Top | Bot
3
4   let rec verifie f (v:bool array) =
5       match f with
6       | X (i) -> v.(i)
7       | Top -> true
8       | Bot -> false
9       | Or (f1, f2) -> (verifie f1 v) || (verifie f2 v)
10      | And (f1, f2) -> (verifie f1 v) && (verifie f2 v)
11      | Not f1 -> not (verifie f1 v)
```

Démonstration

Montrons que **SAT** $\leq_m^{\mathrm{P}}$ **CNF-SAT**

On doit définir une fonction g qui transforme une instance φ de **SAT** en une instance de **CNF-SAT** $g(\varphi)$ telle que :

$$\varphi \text{ satisfiable } \Leftrightarrow g(\varphi) \text{ satifiable}$$

On décrit un algorithme permettant de passer de φ à $g(\varphi)$:

1. On élimine tous les connecteurs $\rightarrow$ et $\leftrightarrow$ en appliquant les équivalences :

$$\begin{cases} \varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2 \\ \varphi_1 \leftrightarrow \varphi_2 (\neg\varphi_1 \vee \varphi_2) \wedge (\neg\varphi_2 \vee \varphi_1) \end{cases}$$

Cela s'effectue en $\mathcal{O}(|\varphi|)$, si on considère que la suppression des $\leftrightarrow$ ne duplique pas réellement les variables φ_1 et φ_2 (avec des pointeurs par exemple).

2. On applique ensuite les lois de De Morgan :

```
let rec morgan phi =
    match phi with
    | Not (And (phi1, phi2)) -> Or (morgan (Not phi1), morgan (Not phi2))
    | Not (Or (phi1, phi2)) -> And (morgan (Not phi1), morgan (Not phi2))
    | Not (Top) -> Bot
    | Not (Bot) -> Top
    | Not (Not phi1) -> morgan phi1
    | And (phi1, phi2) -> And (morgan phi1, morgan phi2)
    | Or (phi1, phi2) -> Or (morgan phi1, morgan phi2)
    | _ -> phi
```

Cela s'effectue en $\mathcal{O}(|\varphi|)$. On obtient à l'issue une forme normale négative ($\neg$ uniquement permis devant des variables seules, et uniquement $\wedge$ et $\vee$).

3. À partir de $\hat{\varphi}$ la formule obtenue à l'étape 2, on construit inductivement $g(\varphi)$. On transforme donc une forme normale négative $f = \hat{\varphi}$ en une forme normale conjonctive F.

- Cas de base : f est un littéral $\top$ ou $\bot$, on pose $F = f$
- Cas inductifs :
  - $f = f_1 \wedge f_2$. On construit F_1 et F_2 à partir de f_1 et f_2. On pose $F = F_1 \wedge F_2$.
  - $f = f_1 \vee f_2$. Ça se complique. On construit inductivement F_1 et F_2, à partir de f_1 et f_2. On pose $F' = (F_1 \vee y) \wedge (F_2 \vee \neg y)$ avec g qui n'apparait pas dans F_1 et F_2.
    F' n'est pass une FNC. On la transforme en F en distribuant les littéraux y et $\neg y$ dans toutes les clauses de F_1 et F_2 (resp.).

Démonstration

On a alors $F = g(\varphi)$. Montrons que f est satisfiable si et seulement si F est satisfiable. Évidemment, on n'a pas directement $F = f$. On pose $\mathcal{V}_F$ les vriables de F, pareil pour$f$. On a $\mathcal{V}_f \subset \mathcal{V}_F$. On va montrer que

- $\text{Mod}(F) \subset \text{Mod}(f)$ si on restreint aux variables de f les modèles.
- $\text{Mod}(f) \subset \text{Mod}(F)$ si on prolonge aux variables de F les modèles.

On procède par induction : LE DEBUT EST MANQUANT

- Cas de base : $F = f$, donc on a v ....
- – $f = f_1 \vee f_2$. On construit encore F via $F' = (F_1 \vee y) \wedge (F_2 \vee \neg y)$ en distribuant. Soit v un modèle de F, c'est un modèle de F'.
  - ∗ Si $v(y) = F$, nécessairement, v est un modèle de F_1 pour satisfaire $(F_1 \vee y)$. Par hypothèse d'induction, $v_{|v_f}$ modèle de f_1. Donc modèle de f.
  - ∗ Si $v(y) = V$, le cas est symétrique.

  Soit v un modèle de f.
  - ∗ si v est un modèle de f_1, alors par hypothèse d'induction, on peut prolonger v en v_1 modèle de F_1. On pose alors $\widetilde{v} : y \mapsto F; x \mapsto v_1(x)$ pour $x \neq y$.
  - ∗ si v est un modèle de f_2, le cas est symétrique.

Il reste à justifier que

Photos du vendredi 07/03 à recopier ici.

Récapulatif

Proposition 11.41

3-SAT est un problème NP-complet

Démonstration

Etape 1 : **3-SAT** est de classe NP (*algo vérificateur* avec une valuation comme *certificat* en temps *polynomial*) (les mots-clés sont en italique)
Etape 2 : montrer que **CNF-SAT** $\leq_m^{\mathrm{P}}$ **3-SAT**.
L'idée est la suivante. Convertir $(l_1 \vee \cdots \vee l_k)$ en $(l_1 \vee l_2 \vee x_{2,3}) \wedge (\neg x_{2,3} \vee l_3 \vee x_{3,4})$.
On définit une fonction g qui prend une formule φ sous FNC (instance de **CNF-SAT**) et construit une instance de **3-SAT** $g(\varphi)$ qui vérifie :

$$\varphi \text{ satisfiable } \iff g(\varphi) \text{ satisfiable}$$

Soit φ une telle FNC, qu'on écrit :

$$\varphi = \bigwedge_{i=1}^{n} C_i$$

Pour $i \in [\![1, n]\!]$, on construit la formule :

$$F_i = \begin{cases} C_i & \text{si } C_i \text{ est de taille au plus 3.} \\ h(C_i) & \text{sinon} \end{cases}$$

où $h(C_i)$, pour $C_i = (l_{1,i} \vee \cdots \vee l_{k,i})$ $(k \geq 4)$, on introduit $k-3$ variables propositionnelles spécifiques à la clause (non déjà présente dans $\mathcal{V}_\varphi$), notées $(x_{j,j+1,i})_{j \in [\![2,\, k-2]\!]}$. Ainsi :

$$F_i = (l_{1,i} \vee l_{2,i} \vee x_{2,3,i}) \wedge (\neg x_{2,3,i} \vee l_{3,i} \vee x_{3,4,i}) \wedge \cdots \wedge (\neg x_{k-2,k-1,i} \vee l_{k-1,i} \vee l_{k,i})$$

Ainsi, $|F_i| = \mathcal{O}(|C_i|)$ (on triple la taille).
On pose alors :

$$g(\varphi) = \bigwedge_{i=1}^{n} F_i$$

Les F_i sont toutes des 3-FNC donc $g(\varphi)$ est une 3-FNC. De plus, $|g(\varphi)| = \mathcal{O}(|\varphi|)$ et est constructible en temps linéaire.

Démonstration

Il reste à montrer que :

$$\varphi \text{ satisfiable } \iff g(\varphi) \text{ satisfiable}$$

- $\boxed{\implies}$: Si φ est satisfiable, soit v un modèle de φ. On souhaite le prolonger sur les nouvelles variables en $\widetilde{v}$ de sorte que $\widetilde{v}$ satisfasse $g(\varphi)$.
  On considère une clause C_i :
  - Si $F_i = C_i$, v convient déjà
  - Si $F_i \neq C_i$, écrivons $C_i = (l_{1,i} \vee \dots l_{k,i})$.

  v satisfait C_i donc v satisfait un littéral $l_{j,i}$ parmi ceux-là. Dans F_i, voir Figure 100.
  On peut prolonger v en posant :

$$\widetilde{v}(x_{p,p+1,i}) = \begin{cases} V & \text{si } p \leq j-1 \\ F & \text{sinon} \end{cases}$$

  $\widetilde{v}$ satisfait bien F_i. On applique cela pour tous les F_i et comme les variables sont spécifiques à une clause, il n'y a pas de contradiction (au sens doublon de variable). $\widetilde{v}$ satisfait $g(\varphi)$. Donc $g(\varphi)$ est satisfiable.

- $\boxed{\impliedby}$: Supposons que $g(\varphi)$ satisfiable.
  Soit v un modèle de $g(\varphi)$. Montrons que v satisfait φ (même s'il est défini sur plus que $\mathcal{V}_\varphi$).
  On considère une clause C_i :
  - Si $F_i = C_i$, v satisfait F_i donc C_i
  - Si $F_i \neq C_i$, notons :

$$\begin{aligned} F_i = &(l_{1,i} \vee l_{2,i} \vee x_{2,3,i}) \wedge \\ &\cdots \wedge \\ &(\neg x_{j,j+1,i} \vee l_{j+1,i} \vee x_{j+1,j+2,i}) \wedge \\ &\cdots \wedge \\ &(\neg x_{k-2,k-1,i} \vee l_{k-1,i} \vee l_{k,i}) \end{aligned}$$

    Supposons par l'absurde que v ne satisfait aucun $l(j,i)$.
    La première clause est satisfaite donc $v(x_{2,3,i}) = V$ nécessairement. Par propagation de clause en clause, on a $v(xj, j+1, i) = V$ pour tout j, et la dernière clause ne peut être satisfaite dans ces conditions. absurde.
    v satisfait donc au moins un littéral l_i donc la clause C_i.

    Au totale, v satisfait toutes les clauses de φ, qui est donc satisfiable.

  Donc g est bien une réduction polynomiale :

$$\textbf{CNF-SAT} \leq_m^{\mathrm{P}} \textbf{3-SAT}$$

  Puis **CNF-SAT** est NP-complet. Ainsi **3-SAT** est NP-difficile.

Au total avec ces deux tirets principaux, on a bien que **3-SAT** est NP-complet.

III.4 Chemin hamiltonien

Définition 11.42 - *chemin hamiltonien*

Dans un graphe $G = (S, A)$, un *chemin hamiltonien* de $s \in S$ à $t \in S$ est un chemin de départ s, d'arrivée t passant exactement une fois par chacun des sommets de S.

Définition 11.43 - *cycle hamiltonien*

Dans un graphe $G = (S, A)$, un *cycle hamiltonien* est un chemin hamiltonien d'origine et d'extrémité égales.

On définit deux problèmes

Définition 11.44 - *Problème* ***CHEMIN-HAM***

Instance : Un graphe $G = (S, A)$ orienté, s et t deux sommets de S. **Question** : Existe-t-il un chemin hamiltonien de s à t ?

Définition 11.45 - *Problème* ***CYCLE-HAM***

Instance : Un graphe $G = (S, A)$ orienté **Question** : Existe-t-il un cycle hamiltonien dans G?

Proposition 11.46 - *concernant ces deux problèmes*

CHEMIN-HAM et **CYCLE-HAM** sont NP-Complets.

Démonstration

Les mots-clés sont en italique.

- Montrons qu'ils sont de classe NP. On choisit comme ensemble des *certificats* les suites des sommets. On peut construire un *algorithme vérificateur* de complexité en la taille de la liste des sommets qui vérifie que chaque sommet du graphe apparaît une et une seule fois (avec un tableau de booléens), qu'il y a bien des arêtes entre deux sommets consécutifs (cela se fait en $\mathcal{O}(1)$ avec des matrices d'adjacence) et éventuellement que l'on a le bon départ, la bonne arrivée.
  Moralement, on vérifie que c'est bien un cycle hamiltonien ou un chemin hamiltonien.
  Pour une instance positive, il existe un bon certificat de taille $|S|$ qui permet de renvoyer Vrai avec l'algorithme vérificateur en *temps polynomial*.
  Pour une instance négative, l'algorihtme renvoie bien sûr toujours faux.

- Montrons maintenant que **3-SAT** $\leq_m^{\mathrm{P}}$ **CYCLE-HAM** et **3-SAT** $\leq_m^{\mathrm{P}}$ **CHEMIN-HAM**.
  Pour φ une 3-FNC, on cherche à construire le graphe G_φ tel que :

$$\varphi \in \textbf{3-SAT}^+ \iff (G_\varphi, s, t) \in \textbf{CHEMIN-HAM}^+$$

  3-SAT$^+$ est l'ensemble des instances positives du problème **3-SAT**.
  De même on cherche un graphe $G_{\varphi'}$ tel que :

$$\varphi \in \textbf{3-SAT}^+ \iff G_{\varphi'} \in \textbf{CYCLE-HAM}^+$$

  G_φ et $G_{\varphi'}$ doivent être de taille polynomiale en $|\varphi|$ et constructible en temps polynomial.
  On pose :

$$\varphi = \bigwedge_{i=1}^{k} C_i$$

  Avec pour $i \in [\![1, k]\!]$, $C_i = (l_{1,i} \vee l_{2,i} \vee l_{3,i})$ On pose le graphe gadget suivant : (Figure 101).
  Ce graphe gadget a la particularité de présenter exactement deux chemins hamiltoniens de s à t :

  - $s \to g \to \cdots \to d \to t$
  - $s \to d \to \cdots \to g \to t$

Démonstration

On pose alors $\mathcal{V}_\varphi = \{x_1, \dots, x_n\}$ l'ensemble des variables qui vont apparaître dans φ.
On construit le graphe $G_{\mathcal{V}_\varphi}$ en utilisant n fois ce graphe gadget : une fois pour chaque variable ! (Voir figure 102).
G_{v_φ} a exactement 2^n chemins hamiltoniens (deux possibilités à chaque fois) de s à t, auxquels on peut faire correspondre une valuation du $\mathcal{V}_\varphi$.
À un chemin hamiltonien de s à t, on fait correspondre une valuation v telle que pour $i[\![1, n]\!]$:

$$v(x_i) = \begin{cases} V & \text{si } g_i \text{ apparaît avant } d_i \text{ dans le chemin hamiltonien} \\ F & \text{sinon} \end{cases}$$

Pour chaque classe C_j, on rajoute un sommet en fonction des cas, comme dans la figure 103.
On obtient un graphe $G_\varphi = (S_\varphi, A_\varphi)$, avec :

$$\begin{cases} |S_\varphi| = \underbrace{k}_{\text{clauses}} + \underbrace{2}_{s,t} + n\underbrace{(2k+4)}_{\text{gadget}} & \text{bien polynomiale en } |\varphi| \\ |A_\varphi| \leq \underbrace{(2k+3)}_{\text{gadget}} 2n + \underbrace{n+1}_{\text{lien entre les gadget}} + \underbrace{6k}_{\text{clauses}} & \text{polynomiale en } |\varphi|. \end{cases}$$

G_φ est bien constructible en temps polynomial.
On construit $G_{\varphi'}$ à partir de G_φ en rajoutant une arête de t à s. **C'est la seule chose qui les sépare, d'où l'intérêt de factoriser les démos** Montrons que :

$$\begin{aligned} & \varphi \in \textbf{3-SAT}^+ \\ \iff & (G_\varphi, s, t) \in \textbf{CHEMIN-HAM}^+ \\ \iff & G_{\varphi'} \in \textbf{CYCLE-HAM}^+ \end{aligned}$$

On doit la dernière équivalence au fait que (t, s) est nécessaire dans le cycle. On montre la première.

- $\boxed{\implies}$: Si φ est satisfiable, soit v un modèle de φ. On s'intéresse au chemin hamiltonien dans G_{v_φ}. Pour chaque clause C_j, on a au moins un littéral satisfait, on transforme le chemin hamiltonien dans G_{v_φ} en chemin hamiltonien dans G_φ en prenant le détour vers C_j correspondant au littéral satisfait.
- Si $(G_\varphi, s, t) \in \textbf{CHEMIN-HAM}^+$, on considère la valuation correspondant au chemin hamiltonien existant. on a au moins un détour pris par clause donc chacune est satisfiable.

Fin à copier
Cours du 14/03

III.5 Réductions classiques

Voir Fig.104

IV Machines de Turing (HP)

Définition 11.47 - *classe P avec machines de Turing*

La *classe P dans modèle de calcul des machines de Turing* correspond aux problèmes de décision que l'on peut résoudre avec une *machine de Turing déterministe* en temps polynomial en la taille de l'instance.

Définition 11.48 - *classse NP avec machines de Turing*

La *classse NP dans le modèle de calcul des machines de Turing* correspond aux problèmes de décision que l'on peut résoudre à l'aide d'une *machine de Turing non déterministe* en temps polynomial en la taille de l'instance.

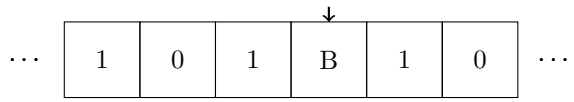
Remarque 11.49 - *implication entre ces classes*

$P \subset NP$ car une *machine de Turing déterministe* est en particulier une *machine de Turing non déterministe*.

Si on sait déterminiser une machine de Turing sans faire exploser (en complexité exponentielle) la taille des calculs, on peut montrer que $P = NP$ (évidemment ce n'est pas encore découvert).

Remarque 11.50 - *introduction aux machines de Turing*

Les machines de Turing constituent une extension de la notion d'automate. Elles utilisent une *bande*infinie sur laquelle elles peuvent lire, écrire, et se déplacer à gauche ou à droite.

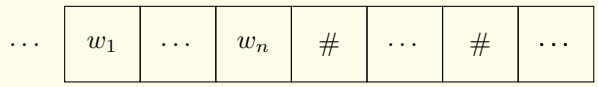


Les machines de Turing constituent une abstraction des ordinateurs tels qu'on les connaît aujourd'hui. Voir Fig 105. On peut aussi se limiter à une demi-droite en guise de bande.

Définition 11.51 - *Machine de Turing*

Une machine de turing est la donnée d'un 7-uplet $\mathcal{M} = (\Sigma, \Gamma, Q, q_0, F, T, \#)$, où :

- Σ est l'*alphabet d'entrée*. C'est l'ensemble fini des lettres que l'on utilise pour écrire le mot donné en entrée à $\mathcal{M}$.



  $w = w_1 \dots w_n \in \Sigma^*$ est l'entrée donnée. On a ici représenté la configuration initiale de la bande pour cette entrée.

- $\#$ est le *symbole blanc*, il sert à remplir les cases non occupées de la bande.
- Γ est l'*alphabet de bande*. Il est fini, et on a :

$$\Sigma \cup \{\#\} \subset \Gamma$$

- Q est un *ensemble fini d'états*
- $q_0 \in Q$ est l'*état initial*.
- $F \subset Q$ est l'*ensemble des états finaux*.
- T est l'*ensemble des transitions*. Il vérifie :

$$T \subset Q \times \Gamma \times Q \times \Gamma \times \{\triangleleft, \triangleright\}$$

  Une transition est alors :

  - soit de la forme $(q_1, a) \to (q_2, b, \triangleright)$, auquel cas on est dans l'état q_1, on lit a, on remplace a par b dans la bande, puis on avance d'une case vers la droite, et on se place en l'état q_2.
  - soit de la forme $(q_1, a) \to (q_2, b, \triangleleft)$, symétriquement.

Exemple 11.52 - *de machine de Turing*

Voir Fig.106

Remarque 11.53 - *concernant les machines de Turing*

La bande est infinie à droite. Lors d'un calcul le nombre de cases occupées (cases sans symbole blanc) peut croître indéfiniment et on peut de fait avoir des calculs infinis. Pour autant, on ne peut pas avoir un nombre infini de cases occupées.

Définition 11.54 - *configuration d'une machine de Turing*

On appelle *configuration d'une machine de Turing* est la donnée de l'état d'une machine de Turing :

- l'état actuel, appelé *état de contrôle*
- le contenu de la bande
- la position de la tête de lecture

On note une $C = (u, q, z) = u\,q\,v$ une telle configuration, où :

- u est le contenu de la bande strictement avant la tête de lecture
- q est l'état courant
- v est le contenu de la bande après la tête de lecture jusqu'à la dernière case occupée incluse. (tous les symboles blancs suivant sont omis)

Exemple 11.55 - *de configuration dans une machine de Turing*

Pour la machine de Turing suivante, Voir fig.107, la configuration représentée est $abca\,q_1\,\#a$

Définition 11.56 - *configuration initale dans une machine de Turing*

La *configuration initale* dans une machine de Turing lorsqu'on lit un mot w sur une machine de Turing $\mathcal{M}$ est :

$$q_0\,w = (\epsilon, q_0, w)$$

Définition 11.57 - *étape de calcul dans une machine de Turing*

Une *étape de calcul* dans une machine de Turing est un couple (C, C') de configurations, noté $C \to C'$ telle que :

- Soit on a $C = uc\,q\,av$ et $C' = u\,q'\,cbv$ et $(q, a) \to (q', b, \triangleleft)$ est une transition, avec :
  - $u, v \in \Gamma^*$
  - $q, q' \in Q$
  - $a, b, c \in \Gamma$
- Soit on a $C = u\,q\,av$ et $C' = ub\,q'\,v$ et $(q, a) \to (q', b, \triangleright)$ est une transition, avec :
  - $u, v \in \Gamma^*$
  - $q, q' \in Q$
  - $a, b \in \Gamma$

Là encore, on appelle *calcul* une suite d'étapes de calcul :

$$C_1 \to \cdots \to C_n$$

Définition 11.58 - *calcul acceptant dans une machine de Turing*

Un *calcul* $C_1 \to \cdots \to C_n$ *acceptant* dans une machine de Turingv vérifie :

- $C_0 = q_0\,w$ une configuration initiale
- $C_k = u\,q\,v$ avec $q \in F$

On dit que $\mathcal{M}$ *accepte* w (et non plus "reconnaît"). On note $\mathcal{L}(\mathcal{M})$ le langage des mots reconnus par $\mathcal{M}$

Remarque 11.59 - *concernant les calculs acceptants*

Si le calcul est amené à sortie de la bande par la gauche (dans la convention où cette bande est une demi-droite), on dit que *le calcule échoue*.

Exemple 11.60 - *de calcul sur une machine de Turing*

En reprenant la précédente machine de Turing, lisant mot $w = a^8$:

$$\begin{aligned}
&q_0\, aaaaaaaa\\
\to &\#\, q_1\, aaaaaaa\\
\to &\#a\, q_2\, aaaaaa\\
\to &\#aA\, q_3\, aaaaa\\
\to &\#aAa\, q_2\, aaaa\\
\to &\#aAaA\, q_3\, aaa\\
\to &\#aAaAa\, q_2\, aa\\
\to &\#aAaAaA\, q_3\, a\\
\to &\#aAaAaAa\, q_2\\
\to &\#aAaAaA\, q_3\, a\\
\to^* &q_4\, aAaAaAa\\
\to &\#\, q_0\, aAaAaAa\\
\to &\#\#\, q_1\, AaAaAa\\
\to &\#\#A\, q_1\, aAaAa\\
\to &\#\#Aa\, q_2\, AaAa
\end{aligned}$$

$$\to^* \#\#A\#AAA\#\#\, q_5$$

Le mot est accepté car le cacul termine. (le tableau deux n'a pas été copié).
On peut montrer que $\mathcal{L}(\mathcal{M}) = \{a^{2^n},\, n \in \mathbb{N}\}$. L'idée est que l'on réalise un "tour" :

$$q_0 \to q_1 \to \begin{cases} q_2 \\ q_3 \end{cases} \to q_4 \to q_0$$

pour elever la moitié des a jusqu'à en avoir un unique.
Si on a un nombre impaire de a, on échoue en q_3 devant un symbole blanc.

Définition 11.61 - *Machine de Turing déterministe*

Une machine de Turing est *déterministe* lorsque :

$$\forall (q,a) \in Q \times \Gamma,\ |\{(q,a,q',c,x) \in T,\ (q',c,x) \in Q \times F \times \{\triangleleft, \triangleright\}\}| \leq 1$$

Remarque 11.62 - *variantes de machines de Turing*

Il existe plusieurs variantes de machines de Turing :

- plusieurs bandes
- bi-infinie

mais leur puissance de calcul est la même.

Définition 11.63 - *langage récursivement énumérable*

Un langage L sur un alphabet Σ est dit *récursivement énumérable* lorsqu'il existe une machine de Turing $\mathcal{M}$ telle que $L = \mathcal{L}(\mathcal{M})$

Définition 11.64

Soit $\mathcal{P}$ un problème de décision, défini par un ensemble I d'instances et une fonction $f : I \to \{vrai, Faux\}$ de prédicat. On peut associer un $\mathcal{P}$ un langage.
Pour $i \in I$ une instance de problème, on note $< i >$ un encodage de i (on suppose que l'encodage est unique à toute instance).
On note $\mathcal{L}_{\mathcal{P}} = \{< i >, i \in I, f(i) = Vrai\}$. C'est le langage des instances positives.

Remarque 11.65

Por résoudre un problème $\mathcal{P}$, on peut utiliser une machine de Turing $\mathcal{M}$ telle que $\mathcal{L}(\mathcal{M}) = \mathcal{L}_{\mathcal{P}}$

Exemple 11.66 - *d'encodages*

Pour les arbres : $\Sigma = \{N, F, (,)\}$.

$$N(FN(FF)N(N(FF)F))$$

est un encodage de l'arbre :

Pour les entiers, $\Sigma = \{0, 1\}$: écriture en base 2. Pour les graphes, nombre de sommets et liste d'arêtes ($\Sigma = \{0, 1, (,), |\}$).

$$((11)|((0|1)|(1|10)))$$

est un encodage de : Voir fig.108

Définition 11.67 - *problème récursivement énumérable*

Un problème de décision $\mathcal{P}$ tel que $\mathcal{L}_{\mathcal{P}}$ est récursivmeent énumérable est dit *récursivement énumérable* : cela correspond aux problèmes *semi-décidables*. Pour une instance positive, on a un calcul fini qui permet de savoir que l'instance est positive.
Pour une instance négative, on peut avoir des calculs infinis : cela ne permet donc pas de conclure.

Définition 11.68 - *langage décidable*

Un langage L écrit sur un alphabet Σ est dit *décidable* lorsqu'il existe une machine de Turing sans calcul infini acceptant L.
On dit alors que la machine *décide le langage* L

Proposition 11.69 - *équivalence entre deux notions*

Poour $\mathcal{P}$ un problème de décision :

$$\mathcal{L}_{\mathcal{P}} \iff \mathcal{P} \text{ décidable}$$

On appelle aussi ces langages des *langages récursifs*.

Proposition 11.70

Si un langage L est tel que :

- L est récursivement énumérable
- $\Sigma^* \setminus L$ est récursivement énumérable

Alors L est récursif.

Remarque 11.71

C'est analogue au fait que si $\mathcal{P}$ et $co\mathcal{P}$ sont semi-décidables, alors $\mathcal{P}$ est décidable.

Définition 11.72 - *encodage d'une machine de Turing*

Soit $\mathcal{M}$ une machine de Turing, $w \in \Sigma^*$. On note :

- $< \mathcal{M} >$ l'encodage de $\mathcal{M}$
- $< w >$ l'encodage de w
- $< \mathcal{M}, w >$ l'encdage de $\mathcal{M}, w$:

$$< \mathcal{M}, \sqcup >=< \mathcal{M} > \$ < w >$$

où \$ est un symbole séparateur tiers.

On note :

$$L_\in = \{< \mathcal{M}, w >,\ w \in \mathcal{L}(\mathcal{M}),\ \mathcal{M} \text{ est une machine de Turing et } w \in \Sigma^*\}$$

$L_\in$ est appelé *langage d'acceptation*. C'est un *langage récursivement énumérable* (ADMIS).

Définition 11.73 - *machine de Turing universelle*

Une machine de Turing $\mathcal{M}_u$ telle que $\mathcal{L}(\mathcal{M}_u) = \mathcal{L}_\in$ existe : on l'appelle *machine universelle*.

Remarque 11.74

Le problème :

- **Instance** : $\mathcal{M}$ machine de Turing, $w \in \Sigma^*$
- **Question** : Est-ce que $\mathcal{M}$ accepte w ?

est un problème semi-décidable. Il est indécidable. Donc $L_\in$ n'est pas récursif. On montre cela comme pour le problème de l'arrêt mais avec des calculs des machines de Turing.

L'existence d'une machine universelle $\mathcal{M}_u$ telle que $\mathcal{L}(\mathcal{M}_u) = \mathcal{L}_\in$ vient du fait que l'on peut simuler le calcul de $\mathcal{M}$ sur w à partir de $< \mathcal{M}, w >$

Définition 11.75 - *mach*

Soit fonction $f : \Sigma^* \to \Gamma^*$ avec Σ et Γ deux alphabets finis et $\Sigma \subset \Gamma$. *f est calculable* lorsqu'il existe une machine de Turing $\mathcal{M}$ qui pour toute entrée $w \in \Sigma^*$ admet un unique calcul acceptant qui s'arrête avec exactement $f(w)$ sur la bande.

Définition 11.76 - *réduction Turing*

Soit A et B deux problèmes de décision et L_A et L_B les langages associés d'alphabets de supports Σ_A et Σ_B. Une *réduction Turing* de A vers B (notée $A \leq_T B$) correspond à une fonction $f : \Sigma_A^* \to \Sigma_B^*$ calculable qui permet de transformer une instance du problème A en une instance du problème B telle que :

$$w \in L_A \longleftrightarrow f(w) \in L_B$$

Remarque 11.77 - *idée*

L'idée est que la machine qui calcule f permet de passer d'une configuration $q_0\, w$ à une configuration $q_0'\, f(w)$. Ensuite, on peut utiliser si elle existe, la machine $\mathcal{M}_B$ telle que $\mathcal{L}(\mathcal{M}) = L_B$ pour savoir si $f(w) = L_B$, donc si $w \in L_A$.

Définition 11.78 - *propriété non triviale*

Une propriété non triviale est une propriété $\mathcal{P}$ telle que :

- il existe L récursivement énumérable qui vérifie $\mathcal{P}$
- et il existe L récursivement énumérable qui ne vérifie pas $\mathcal{P}$.

Théorème 11.79 - *de Rice*

Pour toute propriété *non triviale* $\mathcal{P}$ sur les langages récursivement énumérabbles, le problème de savoir si le langage $\mathcal{L}(\mathcal{M})$ accepté par une machine $\mathcal{M}$ vérifie $\mathcal{P}$ est indécidable.

Démonstration

Soit $\mathcal{P}$ une propriété non triviale sur les langages récursivement énumérables.
On a au moins :

- L_1 récursivement énumérable qui vérifie $\mathcal{P}$
- L_2 récursivement énumérable qui ne vérifie pas $\mathcal{P}$

On pose $L_{\mathcal{P}} = \{< \mathcal{M} >, \mathcal{L}(\mathcal{M}) \text{ vérifie } \mathcal{P}\}$. Montrons que $L_{\mathcal{P}}$ n'est pas un langage décidable en utilisant une réduction faisant intervenir $L_\in$ qui est lui-même indécidable.

On peut écrire $L_\in \leq_T L_{\mathcal{P}}$ pour la réduction Turing concernant les problèmes associés aux langages.
Quitte à considérer $co\mathcal{P}$, on peut supposer sans perte de généralité que $\varnothing$ ne vérifie pas $\mathcal{P}$. Donc $L_1 \neq \varnothing$.
On pose $\mathcal{M}_1$ une machine de Turing telle que $\mathcal{L}(\mathcal{M}_1) = L_1$.
Soit $\mathcal{M}, w$ une instance du problème associée à $L_\in$. ON construit une machine $\mathcal{M}_w$ qui simule l'algorithme suivant : $M_w(u \in \Sigma^*)$: SI $\mathcal{M}$ accepte w: Calculer u sur $\mathcal{M}_1$ (quitte à calculer, rejeter, échouer, ne pas terminer) SINON: Rejeter On admet que l'on peut construire $\mathcal{M}_w$ à partir de $\mathcal{M}$, w et $\mathcal{M}_1$. Montrons que :

$$< \mathcal{M}, w >\in L_\in \longleftrightarrow < \mathcal{M}_w >= L_{\mathcal{P}}$$

- $\boxed{\Longrightarrow}$: Si $< \mathcal{M}, w >\in L_\in$, *i.e.* $\mathcal{M}$ acceptant, on a $\mathcal{L}(\mathcal{M}_w) = L_1$, car on est toujours dans le premier cas de l'algorithme. Or L_1 vérifie $\mathcal{P}$, donc $< \mathcal{M}_w >\in L_{\mathcal{P}}$.
- $\boxed{\Longleftarrow}$: par contraposée.
  Si $< \mathcal{M}, w >\notin L_\in$, *i.e.* $\mathcal{M}$ n'accepte pas w :
  - si le calcul de $\mathcal{M}$ sur w ne termine pas, pour tout u, $\mathcal{M}_w$ ne termine pas, car vérifier si $\mathcal{M}$ accepte w ne termine pas donc $\mathcal{L}(\mathcal{M}_w) = \varnothing$
  - Si $\mathcal{M}$ rejette w, on va dans le "sinon" dans tous lescas et tout mot u est rejeté donc $\mathcal{L}(\mathcal{M}_w) = \varnothing$

  Or $\varnothing$ ne vérifie pas $\mathcal{P}$ donc $< \mathcal{M}_w >\notin L_{\mathcal{P}}$

Définition 11.80 - *complexité, la vraie*

Soit $\mathcal{M}$ une machine de Turing et $\gamma : C_0 \to \dots C_k$ un calcul sur une entrée $w \in \Sigma^*$.
Le *temps de clacul* du calcul γ est :

$$t_{\mathcal{M}}(\gamma) = k$$

L'espace utilisé par le calcul est :

$$s_{\mathcal{M}}(\gamma) = \max_{0 \leq i \leq k} |C_i|$$

Pour une entrée w, la *complexité temporelle* est définie comme :

$$t_{\mathcal{M}}(w) = \min_{\gamma \text{ calcul de } w \text{ sur } \mathcal{M}} \Big(t_{\mathcal{M}}(\gamma) \Big)$$

la *complexité spatiale* est définie par :

$$s_{\mathcal{M}}(w) = \min_{\gamma \text{ calcul de } w \text{ sur } \mathcal{M}} \Big(s_{\mathcal{M}}(\gamma) \Big)$$

Remarque 11.81

On peut définir pour une machine de Turing $\mathcal{M}$, pour tout $n \in \mathbb{N}$:

$$t_{\mathcal{M}}(n) = \max_{|w|=n} \Big(t_{\mathcal{M}}(w) \Big)$$

et :

$$s_{\mathcal{M}}(n) = \max_{|w|=n} \Big(s_{\mathcal{M}}(w) \Big)$$

Définition 11.82 - *classes de complexité*

Soit $f : \mathbb{N} \to \mathbb{R}_+$ (souvent $\mathbb{N}$).

1. $TIME(f(n))$ est l'ensemble des problèmes de décision résolus par une machine de Turing déterministe telle que :
$$t_{\mathcal{M}}(n) = \mathcal{O}\Big(f(n)\Big)$$
2. $NTIME(f(n))$ est l'ensemble des problèmes de décision résolus par une machine de Turing non déterministe telle que :
$$t_{\mathcal{M}}(n) = \mathcal{O}\Big(f(n)\Big)$$
3. On a toujours $TIME(f(n)) \subset NTIME(f(n))$.

$$P = _{k\geq 0}\ TIME(n^k)$$
$$NP = _{k\geq 0}\ NTIME(n^k)$$
$$EXP = EXPTIME = \bigcup_{k\geq 0} TIME(2^{n^k})$$
$$NEXP = NEXPTIME = \bigcup_{k\geq 0} NTIME(2^{n^k})$$

Proposition 11.83 - *machine de Turing*

Pour toute machine de Turing $\mathcal{M}$ non déterministe, il existe une machine $\mathcal{M}'$ déterministe tell que :

$$t_{\mathcal{M}'}(n) = \mathcal{O}(t_{\mathcal{M}}(n)2^{t_{\mathcal{M}}(n)})$$

Remarque 11.84

En pratique, c'est pour cela que les problèmes NP-complets sont résolus en temps EXP.

Proposition 11.85

$P \subset NP \subset EXP \subset NEXP$.