Théorie de la complexité et calcul déterministe

1.1Introduction

- Avec les améliorations continuelles de la puissance de calcul, il serait facile de croire que tout problème de calcul que nous pourrions souhaiter résoudre sera bientôt à portée de main. Malheureusement, cela ne semble pas être le cas.

- Considérez le problème de calcul suivant

- *Le problème du voyageur de commerce.*

- *Problème* **:** étant donné une liste de n villes, $c_1, c_2, \ldots, c_n$ et une matrice symétrique $D_{n \times n}$ de distances, telle que D_{ij} = distance de la ville c_i à la ville c_j
- Déterminez le circuit le plus court optimal en visitant chacune des villes une seule fois.
- Un algorithme naïf évident est : « essayez toutes les visites possibles à tour de rôle et choisissez la plus courte ».
- nous aurions besoin de vérifier n ! Tours
- devront peut-être effectuer au moins 2^n opérations.

- Pour mettre cela en perspective, supposons que nous ayons n = 300 de villes à visiter.

- Si nous pouvions construire un ordinateur utilisant chaque atome de la terre de manière à ce que chaque atome puisse effectuer 10^{10} opérations par seconde et que notre ordinateur ait commencé son calcul à la naissance de la planète, il n'aurait toujours pas fini !

$$\begin{aligned}
\#\text{ seconds in the lifetime of the Earth} &\leq 4.1 \times 10^{17} \\
\#\text{ atoms in the Earth} &\leq 3.6 \times 10^{51} \\
\#\text{ operations performed by our computer} &\leq 1.5 \times 10^{79} \\
2^{300} &\simeq 2 \times 10^{90}.
\end{aligned}$$

- En théorie de la complexité, nous cherchons à classer les problèmes de calcul en fonction de leur difficulté intrinsèque
- Il y a deux questions fondamentales que nous examinerons,
- Un problème est-il intrinsèquement « facile » ou « difficile » à résoudre ?
- Compte tenu de deux problèmes, P1 et P2, lequel est le plus facile à résoudre ?

- Pour montrer qu'un problème est « facile » à résoudre, il suffit de donner un exemple d'algorithme pratique pour sa solution. Cependant, pour montrer qu'un problème est intrinsèquement « difficile », nous devons montrer qu'aucun algorithme pratique de ce type ne peut exister.

- Afin de donner un sens aux questions ci-dessus, nous aurons besoin d'un modèle formel de calcul capturant les propriétés essentielles de tout ordinateur. Le modèle que nous adoptons est la machine déterministe de Turing

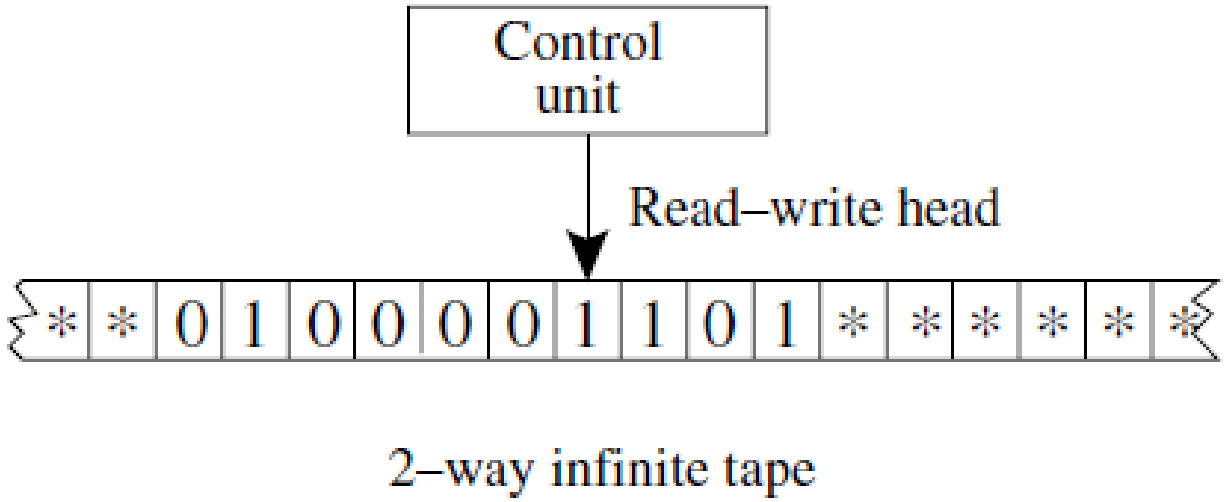

Figure2.1 Une machine de Turing déterministe

2.2 Machine de Turing déterministe

- Une machine de Turing déterministe DTM (Deterministic Turing Machine) se compose de :
- (i) un alphabet fini Σ contenant le symbole vide $*$;
- (ii) une bande infinie bidirectionnelle divisée en carrés, dont l'un est le spécial carré de départ. Chaque carré contient un symbole de l'alphabet Σ. Tout mais un nombre fini de carrés contient le symbole vide spécial $*$, dénotant un carré vide ;
- (iii) une tête de lecture-écriture qui examine un seul carré à la fois et peut se déplacer gauche ($\leftarrow$) ou droite ($\rightarrow$) ;
- (iv) une unité de contrôle avec un ensemble fini d'états Γ comprenant un état de départ, γ_0, et un ensemble d'états d'arrêt.
- Le calcul d'une DTM est contrôlé par une fonction de transition $\delta : \Gamma \times \Sigma \rightarrow \Gamma \times \Sigma \times \{\leftarrow, \rightarrow\}$.

- Au départ, l'unité de contrôle est dans l'état de départ γ_0 et la tête de lecture-écriture balaye le carré de départ. La fonction de transition indique à la machine ce qu'elle doit faire ensuite en fonction du contenu du carré actuel et de l'état actuel de l'unité de contrôle. Par exemple, si l'unité de contrôle est dans l'état γ_{cur} et que le carré courant contient le symbole σ_{cur}, alors la valeur de δ (γ_{cur}, σ_{cur}) indique à la machine trois choses :
- (i) le nouvel état de l'unité de commande (s'il s'agit d'un état d'arrêt, le calcul se termine);
- (ii) le symbole à écrire dans le carré courant ;
- (iii) s'il faut déplacer la tête de lecture-écriture vers la gauche ou la droite d'un carré.

Nous utilisons pour désigner $\Sigma \setminus \{*\}$, l'alphabet des symboles non vides. Nous désignerons la collection de toutes les chaînes finies de Σ_0 par Σ_0^*. Pour $x \in \Sigma_0^*$, nous notons la longueur de x par $| x |$. L'ensemble des chaînes de longueur n à partir de Σ_0^* est noté Σ_0^n.

Le calcul d'une DTM sur l'entrée $x \in \Sigma_0^*$ est simplement le résultat de l'application répétée de la fonction de transition en commençant par x écrit dans le premier $| x |$ carrés de ruban adhésif (ce sont le carré de départ et ceux à sa droite).

Si la machine n'entre jamais dans un état d'arrêt, le calcul ne se termine pas, sinon le calcul se termine lorsqu'un état d'arrêt est atteint. Une seule application de la fonction de transition est appelée une étapes.

Une configuration d'une DTM est une description complète de la machine à un moment particulier d'un calcul : le contenu de la bande, la position de la tête de lecture-écriture et l'état actuel de l'unité de contrôle.

Si une machine DTM s'arrête sur l'entrée $x \in \Sigma^*_0$, le contenu de la bande une fois que la machine s'arrête est appelé la sortie.

On dit qu'une DTM calcule une fonction $f : \Sigma_0^* \rightarrow \Sigma_0^*$ si la machine s'arrête à chaque entrée $x \in \Sigma_0^*$, et la sortie dans chaque cas est f (x).

Le temps sur une DTM est le nombre d étapes que peut prendre la machine avant de s'arrêter.

Example : une machine pour effectuer l'addition d'entiers encodés en unaire est la suivante : (Notez que dans l'algorithme ci-dessous, nous utilisons des abréviations pour réduire le nombre de valeurs de la fonction de transition que nous devons décrire. Par exemple ($\gamma 3$ / $\gamma 4$, 0/1, s, s, ←) est une abréviation pour ($\gamma 3$, 0 , $\gamma 3$, 0, ←), ($\gamma 3$, 1, $\gamma 3$, 1, ←), ($\gamma 4$, 0, $\gamma 4$, 0, ←) et ($\gamma 4$, 1, $\gamma 4$, 1, ←). La lettre s désigne le fait que l'état / symbole reste le même.)

Algorithm *Unary Addition DTM*

The set of states is $\Gamma = \{\gamma_0, \gamma_1, \gamma_2, \gamma_3\}$. The starting state is γ_0 and the only halting state is γ_3. The alphabet is $\Sigma = \{*, 1, +, =\}$.

Input: integers $a, b \geq 0$ in unary with $+, =$. (For example to compute $5 + 2$ we would write '$11111 + 11 =$' on the machine's tape, with the leftmost symbol of the input in the starting square.)

Output: $a + b$ in unary.

$(\gamma_0, 1, \gamma_1, *, \rightarrow)$	# $a \neq 0$, reading a
$(\gamma_0, +, \gamma_2, *, \rightarrow)$	# $a = 0$, erase $+$ read b
$(\gamma_1, 1, \gamma_1, 1, \rightarrow)$	# reading a
$(\gamma_1, +, \gamma_2, 1, \rightarrow)$	# replace $+$ by 1 read b
$(\gamma_2, 1, \gamma_2, 1, \rightarrow)$	# reading b
$(\gamma_2, =, \gamma_3, *, \leftarrow)$	# finished reading b, erase $=$ halt.

Le temps prise par la machine pour effectuer a + b en unaire = nombre d'étapes = a + b + 2.

2.3 Complexité en temps

Si une DTM s'arrête sur l'entrée $x \in \Sigma_0^*$, alors son temps d'exécution sur l'entrée x est le nombre d'étapes que fait la machine lors de son calcul. Nous désignons cela par $t_M(x)$.

On définit la complexité temporelle d'une DTM M qui s'arrête à chaque entrée $x \in \Sigma_0^*$, comme étant la fonction $T_M : N \rightarrow N$ donnée par

$$T_M(n) = \max \{t \mid \text{there exists } x \in \Sigma_0^n \text{ such that } t_M(x) = t\}.$$

Nous considérerons que les temps d'exécution des différents algorithmes sont similaires s'ils ne diffèrent que par un facteur polynomial

Prenons l'exemple de l'addition binaire (TD N°2). Dans notre version de haut niveau, l'algorithme de l'exo1 du td2, le temps d'exécution sur l'entrée $a \geq b$ était O (log a) tandis que pour notre DTM le temps d'exécution était O ($\log^2 a$). Ainsi, pour cet exemple au moins, notre modèle est robuste.

La thèse du temp polynomial de Church–Turing

« Tout algorithme déterministe pratique peut être implémenté en tant que DTM avec un temps d'exécution polynomial ».

2.4 Problèmes de décision et langages

Problèmes de décision : ceux pour lesquels la sortie est vraie ou fausse

Par exemple, le problème de décider si un nombre est premier

PREMIER

Entrée: un entier $n \geq 2$.

Question: n est-il premier?

Nous introduisons un type spécial de DTM qui est particulièrement utile pour examiner de tels problèmes.

DTM accepteurs : Une DTM accepteur est une DTM ordinaire avec exactement deux états d'arrêt : γ_T et γ_F. Ceux-ci doivent être correspondant respectivement à vrai et faux. Une entrée $x \in \Sigma_0^*$ est acceptée par une DTM accepteur si la machine s'arrête à l'état γ_T sur l'entrée x et rejetée si elle s'arrête à l'état γ_F.

Tout ensemble de chaînes $L \subseteq \Sigma_0^*$ est appelé un langage. Si M est une DTM accepteur alors nous définissons le langage accepté par M comme étant

$$L(M) = \{x \in \Sigma_0^* \mid M \text{ accepts } x\}.$$

Si M est une DTM accepteur, L = L (M) et M s'arrête sur toutes les entrées x ∈ , on dit que M décide L

Il existe une correspondance évidente entre les langages acceptées par les DTM accepteurs et les problèmes de décision. Par exemple on peut associer le problème de décision PRIMIER au langage

$$L_{\text{PRIME}} = \{x \mid x \text{ is the binary encoding of a prime number}\}$$

Afin d'obtenir cette correspondance, nous devions choisir un schéma de codage naturel pour l'entrée du problème de décision, dans ce cas binaire.

Pour un problème de décision général Π, nous avons le langage associé

$$L_\Pi = \{x \in \Sigma_0^* \mid x \text{ is a natural encoding of a true instance of } \Pi\}$$

Une DTM accepteur qui décide du langage L, peut être considéré comme un algorithme pour résoudre le problème Π. Étant donné une instance de Π, nous la passons simplement à la machine dans le bon encodage, et retournons la réponse vraie si la machine accepte et faux si elle rejette. Puisque la machine accepte ou rejette toujours, cela donne un algorithme pour le problème Π.

2.5 La classe de complexité « P »

Le but de la théorie de la complexité est de comprendre la difficulté intrinsèque des problèmes de calcul. Lors de l'examen d'un problème de décision, une manière naturelle de mesurer sa difficulté est de considérer la complexité temporelle des machines qui décident du langage associé.

Nous nous intéresserons aux collections de langages qui peuvent tous être décidés par les DTM avec la même limite sur leur complexité temporelle. Une telle collection de langages est appelée une classe de complexité.

Une classe de complexité fondamentale est la classe des langages décidables en temps polynomiaux, ou P. appelé simplement, la classe des langages « traitables ».

$$\mathsf{P} = \{L \subseteq \Sigma_0^* \mid \text{there is a DTM } M \text{ which decides } L \text{ and a polynomial, } p(n) \text{ such that } T_M(n) \leq p(n) \text{ for all } n \geq 1\}.$$

Si est un problème de décision pour lequel $L_\Pi \in P$ on dit qu'il existe un algorithme de temps polynomial pour Π.

Satisfiabilité (SAT) : L'exemple classique d'un problème de décision est la satisfiabilité booléenne. Une fonction booléenne est une fonction $f : \{0, 1\}^n \rightarrow \{0, 1\}$. Nous interprétons «1 » comme vrai et «0 » comme faux.

Les fonctions booléennes de base sont la négation (NOT), la conjonction (AND) et la disjonction (OR). Si x est une variable booléenne alors la négation de x est

$$\overline{x} = \begin{cases} 1, & \text{if } x \text{ is false,} \\ 0, & \text{otherwise.} \end{cases}$$

Un littéral est une variable booléenne ou sa négation. La conjonction d'une collection de littéraux $x_1, \ldots, x_n$ est

$$x_1 \wedge x_2 \cdots \wedge x_n = \begin{cases} 1, & \text{if all of the } x_i \text{ are true,} \\ 0, & \text{otherwise.} \end{cases}$$

La disjonction d'une collection de littéraux $x_1, \ldots, x_n$ est

$$x_1 \vee x_2 \cdots \vee x_n = \begin{cases} 1, & \text{if any of the } x_i \text{ are true,} \\ 0, & \text{otherwise.} \end{cases}$$

Une fonction booléenne, f, est dite sous forme normale conjonctive (ou CNF) si elle s'écrit

$$f(x_1, \ldots, x_n) = \bigwedge_{k=1}^{m} C_k,$$

Où chaque clause, C_k, est une disjonction de littéraux.

$$f(x_1, \ldots, x_6) = (x_1 \vee x_3 \vee \overline{x}_5) \wedge (\overline{x}_4 \vee x_2) \wedge (x_5 \vee x_6),$$
$$g(x_1, \ldots, x_6) = (x_3 \wedge x_5) \vee (x_3 \wedge x_4) \wedge (\overline{x}_6 \wedge x_5) \vee (x_3 \vee x_2).$$

Où, *f* est dans CNF mais *g* ne l'est pas.

Une affectation de vérité pour une fonction booléenne, f (x_1, ..., x_n), est un choix de valeurs $x = (x_1, ..., x_n) \in \{0, 1\}^n$ pour ses variables.
Comme l'assignation de vérité satisfaisante est $x \in \{0, 1\}^n$ tel que f (x) = 1. Si une telle assignation existe, alors f est dite satisfiable.

La satisfiabilité booléenne, également connue sous le nom de SAT, est le problème de décision suivant.

SAT
Entrée: une fonction booléenne, dans CNF
Question: est-ce que f satisfiable?

Nous avons besoin d'un schéma de codage naturel pour ce problème. Nous pouvons utiliser **l'alphabet** = {∗, 0, 1, ∨, ∧, ¬}, codant une variable x_i par la représentation binaire de i. Le littéral x_i peut être encodé en ajoutant un symbole ¬ à l'avant. On peut alors encoder une formule CNF, dans CNF , en utilisant **l'alphabet.**

Par exemple la formule $f(x_1, \ldots, x_5) = (x_1 \vee x_4) \wedge (x_3 \vee \overline{x}_5 \vee x_2) \wedge (\overline{x}_3 \vee x_5),$

Serait codé comme ‘1 ∨ 100 ∧ 11 ∨ ¬101 ∨ 10 ∧ ¬11 ∨ 101’.

Puisqu'aucune clause ne peut contenir plus de 2^n littéraux, la taille d'entrée d'une formule CNF avec n variables et m clauses est O (mn log n).

Un sous-problème important de SAT est le k-SAT, pour $k \geq 1$.

k-SAT
Entrée: une formule booléenne en CNF avec au plus k littéraux dans chaque clause.
Question: est-ce que f satisfiable?

Il est clair que le problème 1-SAT est assez simple. f est satisfiable si et seulement si elle ne contient pas à la fois un littéral et sa négation. Ceci peut clairement être vérifié en temps polynomial et donc $1\text{-SAT} \in P$. Pour $k \geq 2$ la difficulté de k-SAT est moins évidente.

Problèmes de graphe

Les problèmes évidents du monde réel liés aux graphes incluent le problème du voyageur de commerce, les problèmes d'alignement des arbres en génétique et de nombreux problèmes de distribution d'horaire et de planification.

Comme auparavant, nous devons décrire un schéma de codage naturel pour les graphes. Supposons que le graphe que nous souhaitons encoder, G = (V, E), a n sommets et m arêtes. Il existe deux manières évidentes de coder ceci sur une DTM. Nous pourrions utiliser la matrice adjacente, A (G). C'est la matrice symétrique n × n définie par

$$A(G)_{ij} = \begin{cases} 1, & \text{if } \{v_i, v_j\} \in E, \\ 0, & \text{otherwise.} \end{cases}$$

Cette matrice pourrait ensuite être transcrite sous la forme d'une chaîne binaire de longueur n (n + 1) sur la bande de la machine, chaque ligne étant séparée par le symbole &. Avec ce schéma de codage, la taille d'entrée serait O (n^2).

Un problème de décision simple pour les graphes est k-CLIQUE, où $k \geq 2$ est un entier. Il demande si un graphe contient ou non une clique d'ordre k. (C'est une collection de k sommets parmi lesquels toutes les arêtes possibles sont présentes.)

k-CLIQUE
Entrée: un graphe G.
Question: G contient-il une clique d'ordre k?

Si $k \geq 2$ alors **k-CLIQUE $\in$ P**.
Preuve: considérez l'algorithme suivant pour k-CLIQUE.
Entrée: un graphe G = (V, E).
Sortie: vrai si et seulement si G contient une clique d'ordre k.

Algorithme:
pour tout W ⊆ V **tel que** | W | = k
si chaque paire de sommets dans W forme une arête dans E alors la
sortie est vraie
suivant W
sortie false

Nous mesurerons le temps d'exécution de cet algorithme en fonction du nombre d'arêtes présent dans le graphe. Pour un seul ensemble W de taille k, il y a $\binom{k}{2}$ arêtes qui doivent être vérifiées.
Le nombre de possibilités pour l'ensemble W est $\binom{n}{k}$.

Par conséquent, le nombre total d'arêtes vérifiées par l'algorithme est d'au plus $\binom{n}{k}\binom{k}{2}$.
Puisque k est une constante indépendante de l'entrée, le temps d'exécution de cet algorithme est O (n^k) qui est polynomial en n. D'où k-CLIQUE ∈ P.

2.6 Complexité des fonctions :

Bien que nous ayons défini des classes de complexité pour les langages, nous considérerons également la complexité des fonctions. Par exemple, considérons la fonction

$$\text{fac}(n) : \mathbb{N} \to \mathbb{N},$$

$$\text{fac}(n) = \begin{cases} d, & \text{the smallest non-trivial factor of } n \text{ if one exists,} \\ n, & \text{otherwise.} \end{cases}$$

Un algorithme efficace pour calculer fac(n) briserait un bon nombre des crypto-systèmes les plus couramment utilisés. Pour cette raison, la détermination de la complexité de cette fonction est un problème extrêmement important.

La classe des fonctions traitables FP, l'analogue de la classe P des langages traitables, est

$$\text{FP} = \{f : \Sigma_0^* \to \Sigma_0^* \mid \text{there is a DTM } M \text{ that computes } f \text{ and a polynomial } p(n) \text{ such that } T_M(n) \leq p(n) \text{ for all } n \geq 1\}.$$

Si f ∈ FP alors on dit que f est calculable en temps polynomial.
Un exemple que nous avons déjà vu est l'addition d'entiers binaires qui est une fonction dans FP.

2.7 Complexité spatiale

Si une DTM s'arrête sur l'entrée $x \in \Sigma_0^*$, alors l'espace utilisé sur l'entrée x est le nombre de carrés de bande distincts examinés par la tête de lecture-écriture de la machine lors de son calcul. Nous désignons cela par S_M (x).

Si M est une DTM qui s'arrête pour chaque entrée $x \in \Sigma_0^*$, alors la complexité spatiale de M est la fonction $S_M : N \rightarrow N$ définie par

$$S_M(n) = \max \{s \mid \text{there exists } x \in \Sigma_0^n \text{ such that } s_M(x) = s\}$$

La classe de complexité spatiale la plus importante est la classe des langages qui peuvent être décidés dans l'espace polynomial,

$$\text{PSPACE} = \{L \subseteq \Sigma_0^* \mid \text{there is a DTM } M \text{ which decides } L \text{ and a polynomial, } p(n), \text{ such that } S_M(n) \leq p(n) \text{ for all } n \geq 1\}$$

Il est clair que l'espace est une ressource plus précieuse que le temps dans le sens où la quantité d'espace utilisée dans un calcul est toujours limitée au-dessus par le temps que prend le calcul.

Si un langage L peut être décidé en temps f(n) alors L peut être décidé dans l'espace f(n).

Preuve : Le nombre de carrés examinés par la tête de lecture-écriture d'une DTM ne peut pas être supérieur au nombre d'étapes qu'il prend.

P ⊆ PSPACE

Une autre classe de complexité temporelle importante est la classe de langage décidable en temps exponentiel

$$\mathsf{EXP} = \{L \subseteq \Sigma_0^* \mid \text{there is a DTM } M \text{ which decides } L \text{ and a polynomial, } p(n), \text{ such that } T_M(n) \leq 2^{p(n)} \text{ for all } n \geq 1\}$$

Bien que l'espace puisse être plus précieux que le temps, étant donné un temps exponentiel, nous pouvons calculer tout ce qui peut être calculé dans l'espace polynomial.

P ⊆ PSPACE ⊆ EXP