Calculabilité et Complexité (11X008) - Printemps 2026

7. Non-déterminisme

Enseignant: Arnaud Casteigts

Assistants: M. De Francesco, M. Marseloo
Moniteurs: E. Bussod, N. Beghdadi

Dans ce cours, nous rappelons le fonctionnement des machines de Turing non-déterministes et nous présentons les classes de complexité NTIME et NSPACE pour ces machines. Nous discutons ensuite des relations entre ces classes et leurs versions déterministes. La relation entre TIME et NTIME est encore mal comprise, notamment le cas particulier de "P vs NP", qui est une question centrale. Pour l'espace, c'est beaucoup plus clair, nous présentons notamment le *théorème de Savitch*, qui établit que NSPACE $(f(n)) \subseteq$ SPACE $(f(n)^2)$.

7.1 Machine de Turing non-déterministe

Pour rappel, à tout moment, une machine se trouve dans une certaine **configuration** qui représente son état dans l'automate, la position de ses têtes de lecture, et le contenu de ses bandes. Quand elle effectue une transition, elle passe d'une configuration à une autre. Dans le cas d'une machine **déterministe**, il n'y a qu'une seule transition possible à chaque étape, l'exécution prend donc la forme d'un **chemin** de configurations (Figure 1 à gauche). À l'inverse, une machine **non-déterministe** peut avoir plusieurs transitions possibles à chaque étape, son exécution prend donc la forme d'un **arbre** (Figure 1 à droite).

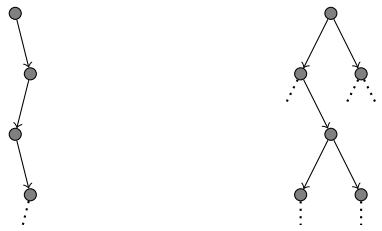
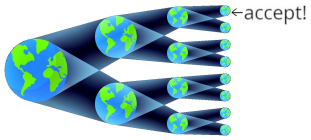


FIGURE 1 – Exécution d'une machine déterministe (à gauche) ou non-déterministe (à droite).

L'interprétation que l'on a des machines non-déterministes est qu'à chaque fois qu'elles ont plusieurs choix, elles les effectuent tous "en parallèle" dans des univers dictincts. La machine accepte si *au moins* une des branches accepte.



Notez que le nombre de choix à chaque étape est limité : il n'y a que $O(1)$ transitions possibles à partir d'une configuration, car la machine et elle-même a un nombre fini d'états et de symboles dans l'alphabet.

D'un point de vue physique, les machines (ou algorithmes) non-déterministes n'existent pas réellement, elles nous offrent surtout un cadre conceptuel qui s'est avéré fertile jusqu'à présent. En particulier, il existe une autre manière d'interpréter le non-déterminisme, qui correspond à des choses bien réelles, dont nous parlerons plus tard.

7.2 NTIME et NSPACE

Comment mesurer le temps ou l'espace utilisés par une machine non-déterministe, dans la mesure où elle a plusieurs branches d'exécution ? C'est très simple : le temps d'exécution correspond à la longueur de la branche la plus longue. De même, l'espace utilisé se mesure sur la branche qui en utilise le plus. On peut maintenant définir les analogues de TIME et SPACE pour les machines non-déterministes :

- NTIME($f(n)$) : Langages décidables en **temps** $O(f(n))$ par une machine de Turing *non-déterministe* (sans se soucier de l'espace).
- NSPACE($f(n)$) : Langages décidables en **espace** $O(f(n))$ par une machine de Turing *non-déterministe* (sans se soucier du temps).

Pour éviter les ambiguités, les classes TIME et SPACE sont parfois appelées DTIME et DSPACE. (Nous ne le ferons pas.) Les versions non-déterministes des classes LOGSPACE, P, PSPACE, et EXP sont naturellement NLOGSPACE, NP, NPSPACE, et NEXP.

Dans cette liste, vous reconnaîtrez certainement la fameuse classe NP, qui correspond aux langages qu'on peut décider en temps polynomial avec une machine non-déterministe.

7.2.1 Théorèmes de hiérarchie

Comme pour les classes TIME et SPACE, il existe pour la classe NTIME un théorème de hiérarchie en temps et pour la classe NSPACE un théorème de hiérarchie en espace : plus de temps permet de résoudre plus de problèmes ; plus d'espace permet de résoudre plus de problèmes. Nous ne nous attarderons pas dessus, mais sachez-le.

7.3 Relation entre déterminisme et non-déterminisme

Comparer les classes déterministes et non-déterministes est plus intéressant. Notez que les machines déterministes sont un cas particulier de machine non-déterministe (avoir un seul

choix est un cas particulier d'en avoir plusieurs), nous avons donc directement les inclusions suivantes pour tout $f(n)$:

- $\mathsf{TIME}(f(n)) \subseteq \mathsf{NTIME}(f(n))$
- $\mathsf{SPACE}(f(n)) \subseteq \mathsf{NSPACE}(f(n))$

Ces inclusions sont triviales, peut-on dire d'autres choses ?

7.3.1 Relations entre TIME et NTIME

L'impact du non-déterminisme sur le *temps* est une grande question en informatique théorique. En particulier, ce sujet encapsule des questions très profondes qui ferons l'objet de plusieurs cours suivants. La principale question est certainement celle de P versus NP. Autrement dit, les machines non-déterministes peuvent-elles décider, en temps polynomial, plus de langages que les machine déterministes ? Cette question est d'autant plus importante que de nombreux problèmes réels sont dans NP. Si $\mathsf{P} = \mathsf{NP}$, alors tous ces problèmes peuvent être résolus rapidement par nos ordinateurs. Cependant, on soupçonne que ce n'est pas le cas et que les machines déterministes sont exponentiellement plus lentes.

7.3.2 Relations entre SPACE et NSPACE

Pour l'espace, on comprend mieux la relation. Le **théorème de Savitch** établit que $\mathsf{NSPACE}(f(n)) \subseteq \mathsf{SPACE}(f(n)^2)$. Autrement dit, tout langage décidable par une machine non-déterministe peut aussi être décidé par une machine déterministe qui n'utilise "que" quadratiquement plus d'espace. On est donc loin du gain exponentiel suspecté pour le temps.

En combinant la relation mentionnée plus haut avec le théorème de Savitch, on obtient un encadrement assez précis :

$$\mathsf{SPACE}(f(n)) \subseteq \mathsf{NSPACE}(f(n)) \subseteq \mathsf{SPACE}(f(n)^2)$$

Ces inclusions impliquent $\mathsf{PSPACE} = \mathsf{NPSPACE}$, car un surcoût quadratique à une quantité polynomiale reste une quantité polynomiale. Par ailleurs, on peut montrer que $\mathsf{NLOGSPACE} \subseteq \mathsf{P}$. On sait aussi que $\mathsf{NP} \subseteq \mathsf{PSPACE}$ (toute machine non-déterministe en temps polynomial peut être simulée par une machine déterministe utilisant une quantité d'espace polynomiale). Enfin, on sait que $\mathsf{NSPACE} \subseteq \mathsf{EXP}$. Pour résumer, l'état actuel des connaissances est :

$$\mathsf{LOGSPACE} \subseteq \mathsf{NLOGSPACE} \subseteq \mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} = \mathsf{NPSPACE} \subseteq \mathsf{EXP} \subseteq \mathsf{NEXP}$$

Le plus frappant est que pour chaque inclusion $\subseteq$ dans cette expression, on ne sait pas si l'inclusion est stricte : chaque paire de classes consécutives est potentiellement égale !

7.4 Théorème de Savitch

Pour tout $f(n) \geq \log n$,

$$\mathsf{NSPACE}(f(n)) \subseteq \mathsf{SPACE}(f(n)^2)$$

La preuve n'est pas triviale, mais elle est algorithmiquement intéressante.

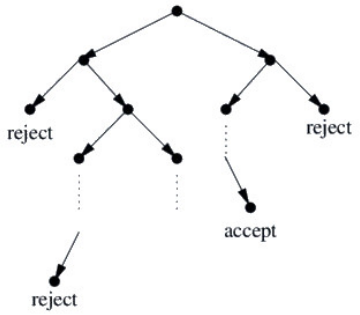
7.4.1 Principe général

Pour simplifier les notations, nous allons prendre le cas particulier de $f(n) = n$ et montrer que si $L \in \mathsf{NSPACE}(n)$ alors $L \in \mathsf{SPACE}(n^2)$. (Les arguments utilisés sont généraux.)

Soit $L \in \mathsf{NSPACE}(n)$. Par définition, il existe une machine *non-déterministe* M qui décide L en utilisant $O(n)$ espace. L'objectif est de montrer qu'il existe une autre machine M' qui est *déterministe* et qui décide L en utilisant $O(n^2)$ espace.

Puisque M existe, on peut supposer que notre machine M' connaît la description $\langle M \rangle$. Notre machine M' ne va pas simuler M à proprement parler. Elle va utiliser $\langle M \rangle$ pour savoir si M accepterait un mot w donné.

Notez que M utilise $O(n)$ espace, elle a donc $2^{O(n)}$ configurations possibles[1] et ses branches d'exécution ont donc aussi une profondeur bornée par $2^{O(n)}$. Vous pouvez imaginer quelque chose comme ça :



Notre objectif est de déterminer si au moins l'une de ces branches accepte. Bien évidemment, on pourrait simuler M en effectuant implicitement un parcours en profondeur de cet arbre à la recherche d'une configuration acceptante. Le problème est que c'est trop coûteux en espace, car on doit mémoriser les configurations d'où l'on vient pour pouvoir remonter, sachant que les branches peuvent avoir une profondeur de $2^{O(n)}$. On voudrait réussir à décider cela en utilisant seulement $O(n^2)$ espace. Est-ce possible ?

1. Notez que $3^{O(n)} = 2^{O(n)}$. C'est vrai pour n'importe quelle base constante, en faisant varier la constante dans le $O(n)$ en exposant.

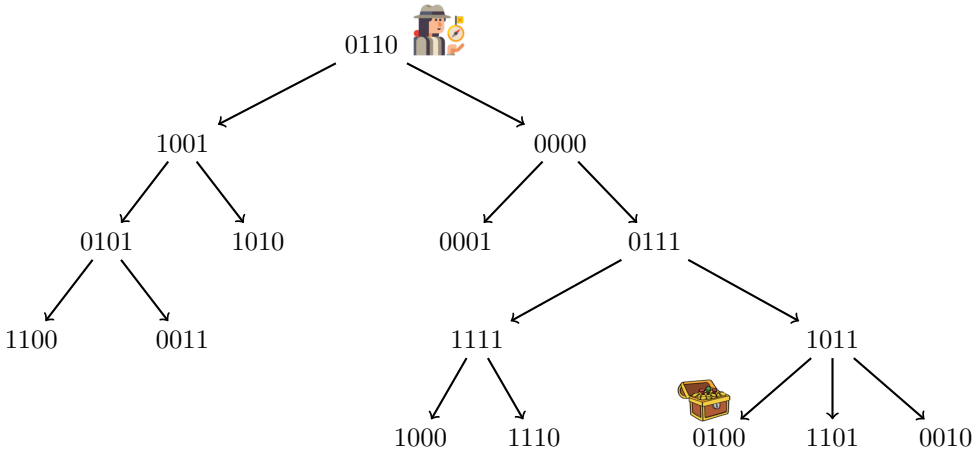
Oui, c'est possible, bien qu'effroyablement lent. La première étape est d'énumérer toutes les configurations possibles de M, même celles qui ne seront jamais atteintes pendant l'exécution. Puis, pour chacune, on essaie de déterminer s'il existe une branche d'exécution qui peut l'atteindre depuis la configuraiton initiale. Si oui, c'est gagné, cela veut dire que M accepte. Sinon, cela veut dire que M rejette.

Bien sûr, cela prend très longtemps : chaque configuration a une taille de $O(n)$, il y en a donc $2^{O(n)}$ à énumérer. Mais c'est OK, seul l'espace nous importe ici, et on peut recycler l'espace au fur et à mesure. L'algorithme général est celui-ci :

```
Pour chaque configuration possible:
|  Si cette configuration est acceptante:
|  |  S'il existe un chemin de la conf initiale vers cette configuration:
|  |  |  Accepter
Rejeter
```

7.4.2 Cœur de la preuve

Maintenant qu'on a une idée générale, le problème est le suivant : étant donné la description d'une machine non-déterministe $\langle M \rangle$ et deux de ses configurations c_1 et c_2 (chacune de taille $O(n)$), comment décider s'il existe un chemin de c_1 à c_2 dans l'arbre d'exécution de $\langle M \rangle$, en utilisant seulement $O(n^2)$ espace ? Vous pouvez imaginer cela comme une chasse au trésor :



Notez que la longueur maximale d'un chemin de c_1 à c_2, si un tel chemin existe, est de $L = 2^{O(n)}$. Par ailleurs, si un tel chemin existe, alors il existe une configuration intermédiaire c_{mid} telle que la longueur de c_1 à c_{mid} est au plus de $L/2$ et idem de c_{mid} à c_2. L'algorithme consiste à énumérer toutes les configurations possibles pouvant jouer le rôle de c_{mid}, et de demander pour chacune s'il existe un chemin de c_1 à c_{mid} de longueur $\leq L/2$ et un chemin de c_{mid} à c_2 de longueur $\leq L/2$. Ce problème est récursif : Pour savoir s'il existe un tel chemin

de c_1 à c_{mid}, on peut énumérer toutes les configurations c_{mid_mid} en demandant s'il existe un chemin de c à c_{mid_mid} et un chemin de c_{mid_mid} à c_{mid}, tous deux de longueur $L/4$, etc. La récursion s'arrête si la longueur demandée est de 0 (on répond oui) ou 1 (on répond oui s'il existe une transition directe de la première vers la seconde). Voici l'algorithme :

```
def can_reach(c1,c2,L):
   if L==0:
      return (c1==c2)
   else if L==1:
      return (M has a transition from c1 to c2)
   else:
      for each configuration c_mid:
         if can_reach(c1, c_mid, ⌊L/2⌋) and can_reach(c_mid, c2, ⌈L/2⌉):
            return True
      return False
```

Cet algorithme est horriblement lent, mais ce n'est pas grave, nous ne sommes pas pressés ! Ce qui est remarquable, c'est que l'espace qu'il utilise est tout petit. À chaque fois que l'on récurse, on peut effacer quasiment tout notre carnet. Les seules informations mémorisées à un instant donné sont les configurations intermédiaires directement impliquées dans la récursion. La longueur du chemin étant divisée par deux à chaque étape, on ne mémorise donc qu'un nombre de configurations logarithmique dans la longueur initiale $L = 2^{O(n)}$ du chemin, ce qui fait donc $\log 2^{O(n)} = O(n)$ configurations à stocker à tout moment. Chaque configuration ayant une taille de $O(n)$ elle-même, cela donne bien $O(n^2)$ espace utilisé.