Calculabilité et Complexité (11X008) - Printemps 2025

7. Non-déterminisme

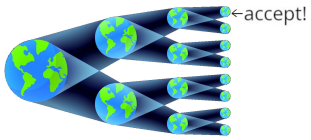
Enseignant: Arnaud Casteigts

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

Dans ce cours, nous passons en revue les classes génériques pour les machines/algorithmes non-déterministes, à savoir NTIME et NSPACE. Nous discutons certaines relations entre ces classes et les classes déterministes TIME et SPACE que nous connaissons. Le résultat principal ici est le *théorème de Savitch*, qui établit une relation forte entre SPACE et NSPACE.

7.1 Machine de Turing non-déterministe

Pour rappel, une **machine de Turing non-déterministe** fonctionne comme une machine déterministe, à ceci près qu'il peut y avoir plusieurs transitions possibles depuis un même état et un même symbole pointé par la tête de lecture (ou symboles pointés, pour des machines à plusieurs bandes). Autrement dit, la machine peut avoir plusieurs choix possibles. Dans ce cas, chacun des choix est effectué en parallèle, la machine se dupliquant dans des univers séparés et continuant son exécution indépendamment des autres branches. Une telle machine accepte le mot d'entrée si et seulement si *au moins* une des branches accepte.



Intuitivement, vous pouvez imaginer un algorithme qui peut explorer plusieurs choix simultanément, et ce, de manière récursive. En revanche, le nombre de choix à chaque étape doit rester *constant*. L'exécution d'un tel algorithme/machine définit un **arbre d'exécution**, dont la racine est la configuration initiale de la machine.

Les machines non-déterministes *n'existent pas*, il ne s'agit donc pas d'un modèle réaliste. Il est cependant possible d'interpréter les résultats de manière déterministe (nous y reviendrons). Son étude est donc centrale en complexité algorithmique.

7.2 NTIME et NSPACE

Le temps d'exécution d'une machine non-déterministe correspond au temps utilisé par la branche *la plus longue*. De même, l'espace utilisé correspond à la mémoire utilisée par la

branche qui *utilise le plus* de mémoire. 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.) La version non-déterministe des classes LOGSPACE, P, PSPACE, et EXP vues dans les cours précédents se définit de la même manière, mais en utilisant NTIME et NSPACE. Cela donne les classes NLOGSPACE, NP, NPSPACE, et NEXP.

Il existe pour le non-déterminisme des théorèmes de hiérarchie en temps et de hiérarchie en espace similaires à ceux vus dans le cours précédent : plus de temps permet de résoudre plus de problèmes ; plus d'espace permet de résoudre plus de problèmes. Sachez-le.

7.3 Relation entre déterminisme et non-déterminisme

Les machines déterministes sont un cas particulier de machine non-déterministe, nous avons donc directement les inclusions suivantes pour tout $f(n)$:

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

En effet, avoir un seul choix possible est un cas particulier d'en avoir plusieurs. En fait, une machine déterministe peut être vue comme une machine non-déterministe dont l'arbre d'exécution est un simple chemin (sans branchement).

7.3.1 Relations entre TIME et NTIME

L'impact du non-déterminisme sur le *temps* est encore mal compris. On suspecte le non-déterminisme d'être *exponentiellement* plus rapide que le déterminisme, mais personne n'arrive à le démontrer. En fait, ce sujet encapsule certaines des questions les plus profondes en informatique, qui ferons l'objet de plusieurs cours suivants, notamment sur les relations entre P (temps polynomial déterministe) et NP (temps polynomial non-déterministe).

7.3.2 Relations entre SPACE et NSPACE

Pour l'espace, on comprend beaucoup mieux la relation. Notamment, le **théorème de Savitch** (présenté plus bas) établit que pour toute $f(n) \geq n$, NSPACE($f(n)$) $\subseteq$ 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, entre autres, que $\mathsf{PSPACE} = \mathsf{NPSPACE}$ (espace polynomial déterministe = espace polynomial non-déterministe). En effet, un surcoût quadratique à une quantité polynomiale reste une quantité polynomiale.

7.4 Théorème de Savitch

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

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

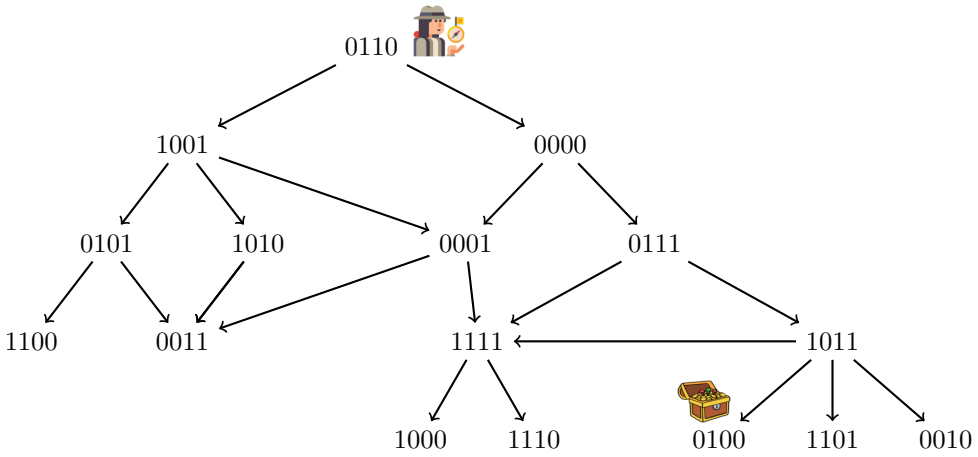
Par souci pédagogique, nous allons découper la preuve en deux morceaux disjoints, le premier consistant en une chasse au trésor.

7.4.1 Chasse au trésor

Voici le problème : Nous nous trouvons initialement sur un sommet d'un graphe orienté G à n sommets et sans cycles. Chaque sommet a un identifiant unique qui est un mot binaire de taille $\log n$. Tous les identifiants sont utilisés par exactement un sommet. Il y a un trésor caché sur un sommet de ce graphe, dont on connaît l'identifiant, mais on ne sait pas s'il existe un chemin depuis notre point de départ vers ce trésor (il se peut qu'il n'y en ait pas). Le but est de décider si un tel chemin existe.

À chaque étape, nous pouvons déterminer l'identifiant des successeurs du sommet courant et nous téléporter sur n'importe quel sommet (successeur ou non) en spécifiant son identifiant. Enfin, nous avons un petit carnet sur lequel nous pouvons écrire et effacer des informations pour accomplir notre mission.

Algorithme naïf : Effectuer un parcours en profondeur de ce graphe : À chaque étape, on se déplace sur un successeur du sommet courant, en notant sur le carnet le sommet duquel on vient (prédecesseur) et les sommets déjà visités. S'il n'y a plus de successeur non-visité, on retourne sur le prédécesseur et on continue. Si un chemin existe entre notre point de départ et le trésor, alors cette stratégie garantit qu'on va le trouver.



Malheureusement, cette stratégie est susceptible de nécessiter le stockage de $O(n)$ identifiants (p.ex. s'il y a un très long chemin), donc $O(n \log n)$ espace sur notre carnet, qui est trop petit ! Je ne vous l'avais pas dit, mais notre carnet n'a que $o(n)$ espace, il ne peut pas contenir tous les identifiants nécessaires :-(. Peut-on quand même résoudre ce problème ?

Algorithme futé : Let problème est de savoir s'il existe un chemin depuis le sommet de départ s vers le trésor t. Si c'est le cas, alors ce chemin doit avoir une longueur $\leq n$. Il existe donc forcément un sommet intermédiaire s_{mid} sur ce chemin telle que la longueur de s à s_{mid} est au plus de $n/2$ et la longueur de s_{mid} à t est au plus de $n/2$. La stratégie est d'énumérer tous les sommets s_{mid} possibles, en se demandant pour chacun s'il existe un chemin de s à s_{mid} et de s_{mid} à t, tous deux de longueur au plus $n/2$. Ce problème est récursif : Pour savoir s'il existe un tel chemin de s à s_{mid}, on peut énumérer tous les sommets s_{mid_mid} en se demandant s'il existe un chemin de s à s_{mid_mid} et un chemin de s_{mid_mid} à s_{mid}, tous deux de longueur $n/4$ et faire pareil de l'autre côté. La récursion s'arrête quand la longueur demandée est de 0 ou 1. Voici le code :

```
def can_reach(s,t,n):
   if n==0:
      return (s==t)
   else if n==1:
      return (t in successors(s))
   else:
      for each identifier s_mid:
         path_1_OK = can_reach(s, s_mid, ⌊n/2⌋)
         path_2_OK = can_reach(s_mid, t, ⌈n/2⌉)
      return (path_1_OK and path_2_OK)
```

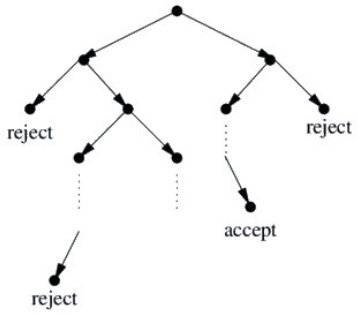
Cet algorithme est horriblement lent, mais ce n'est pas grave, nous ne sommes pas pressés ! Ce qui est remarquable, c'est que l'espace qu'il utilise sur notre carnet est tout petit. À chaque

fois que l'on récurse, on peut effacer quasiment tout notre carnet. Les seules informations réellement mémorisées sont les sommets donnés en entrée dans les étapes *plus hautes* de la récursion. La longueur du chemin étant divisée par deux à chaque étape, on ne mémorise donc à un moment donné que $\log n$ identifiants, chacun de taille $\log n$, soit un total de $(\log n)^2$ espace.

7.4.2 La preuve de Savitch

Supposons qu'un langage L peut être décidé par une machine *non-déterministe* M qui utilise $f(n)$ espace. La description $\langle M \rangle$ de cette machine existe et nous savons que la machine utilise $f(n)$ espace. Nous allons utiliser ces informations pour montrer qu'il existe aussi une machine *déterministe* M' qui décide le même langage en espace $O(f(n)^2)$. Notre machine M' va simuler M (dans un sens nouveau).

Réfléchissons à l'exécution de la machine M. Tout d'abord, cette machine utilise un espace $f(n)$, donc elle a essentiellement $2^{f(n)}$ configurations possibles. À partir de sa configuration initiale, elle va explorer (de manière non-déterministe) un sous-ensemble de ces configurations, ce qui peut être représenté par un arbre d'exécution comme celui-ci :



Si l'on comprend que plusieurs configurations de cet arbre sont susceptibles d'être les mêmes, on a en fait un graphe orienté similaire à celui de la chasse au trésor (sans cycles également, car on suppose que M décide L, donc aucune de ses branches ne boucle). L'objectif de notre nouvelle machine M' est de déterminer, de manière *déterministe*, si ce graphe d'exécution possède un chemin menant à une configuration acceptante pour une entrée donnée. Si un tel chemin existe, notre machine déterministe M' acceptera ; sinon, elle rejettera. Elle répondra ainsi la même chose que M est décidera donc L. C'est exactement le problème de la chasse au trésor, où les identifiants des sommets correspondent aux configurations et où il est possible de déterminer les successeurs d'une configuration (en utilisant $\langle M \rangle$ pour déterminer les transitions possibles). La seule différence est que l'on ne sait pas quelles configurations de M sont acceptantes. Il suffit pour cela d'ajouter une boucle supplémentaire qui énumère une à une toutes les configurations possibles de M, vérifie si elles sont acceptantes (c'est facile) et si oui, lance ensuite l'algorithme de la chasse au trésor où le trésor est

cette configuration acceptante. L'espace peut bien sûr être recyclé entre deux appels de l'algorithme, c'est donc gratuit en espace. Si la machine M a un au moins un chemin acceptant, notre stratégie le trouvera nécessairement.

Complexité : Le problème de la chasse au trésor utilise $(\log n)^2$ espace pour un graphe à n sommets. Ici, nous avons $2^{f(n)}$ sommets, donc on a besoin de $O((\log(2^{f(n)}))^2) = O(f(n)^2)$ espace. □